Kostenloses eBook

LERNEN F#

Free unaffiliated eBook created from
Stack Overflow contributors.

#f#

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fsharp

It is an unofficial and free F# ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official F#.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit F

Bemerkungen

F # ist eine "funktional zuerst" Sprache. Sie können alle verschiedenen Arten von Ausdrücken sowie Funktionen kennenlernen .

Der F # -Compiler - Open Source - kompiliert Ihre Programme in IL, was bedeutet, dass Sie F # -Code aus jeder .NET-kompatiblen Sprache wie C # verwenden können . und führen Sie es unter Mono, .NET Core oder .NET Framework unter Windows aus.

Versionen

Ausführung	Veröffentlichungsdatum
1.x	2005-05-01
2,0	2010-04-01
3,0	2012-08-01
3.1	2013-10-01
4,0	2015-07-01

Examples

Installation oder Setup

Windows

Wenn Sie Visual Studio (eine beliebige Version einschließlich Express und Community) installiert haben, sollte F # bereits enthalten sein. Wählen Sie einfach F # als Sprache, wenn Sie ein neues Projekt erstellen. Oder besuchen Sie http://fsharp.org/use/windows/ für weitere Optionen.

OS X

Xamarin Studio unterstützt F #. Alternativ können Sie VS Code für OS X verwenden , einen plattformübergreifenden Editor von Microsoft.
Wenn Sie mit der Installation von VS Code fertig sind, starten Sie `VS Code Quick Open` (`Strg` + `P`) und führen Sie die `ext install Ionide-fsharp`

Sie können auch Visual Studio für Mac in Betracht ziehen.

Es gibt andere Alternativen, die hier beschrieben werden .

Linux

Installieren Sie die Pakete `mono-complete` und `fsharp` über den Paketmanager Ihrer Distribution (Apt, Yum usw.). Für eine gute Bearbeitung Erfahrung, verwenden Sie entweder Visual Studio - `ionide-fsharp` `ionide-installer` Code und installieren Sie die `ionide-fsharp` Plugin, oder verwenden Atom und die Installation von `ionide-installer` - Plugin. Weitere Optionen finden Sie unter http://fsharp.org/use/linux/ .

Hallo Welt!

Dies ist der Code für ein einfaches Konsolenprojekt, das "Hallo, Welt!" zu STDOUT und beendet den Exit-Code `0`

```
[<EntryPoint>]
let main argv =
    printfn "Hello, World!"
    0
```

Beispiel für eine Aufteilung Zeile für Zeile:

- `[<EntryPoint>]` - Ein .net-Attribut , das "die Methode, mit der Sie den Einstiegspunkt festlegen" Ihres Programms (Quelle) kennzeichnet.
- `let main argv` - definiert eine Funktion namens `main` mit einem einzigen Parameter `argv` . Da dies der Programmeinstiegspunkt ist, wird `argv` ein Array von Strings sein. Der Inhalt des Arrays sind die Argumente, die an das Programm übergeben wurden, als es ausgeführt wurde.
- `printfn "Hello, World!"` - Die Funktion `printfn` gibt den übergebenen String ** als erstes Argument aus und fügt auch eine neue Zeile hinzu.
- `0` - F # -Funktionen geben immer einen Wert zurück, und der zurückgegebene Wert ist das Ergebnis des letzten Ausdrucks in der Funktion. Wenn Sie als letzte Zeile `0` eingeben, gibt die Funktion immer Null (eine ganze Zahl) zurück.

** Dies ist eigentlich *keine* Zeichenfolge, obwohl es wie eine aussieht. Es ist eigentlich ein TextWriterFormat , das optional die Verwendung von statisch typgeprüften Argumenten erlaubt. Aber für ein "Hallo Welt" -Beispiel kann man es als eine Zeichenkette betrachten.

F # interaktiv

F # Interactive ist eine REPL-Umgebung, in der Sie F # -Code Zeile für Zeile ausführen können.

Wenn Sie Visual Studio mit F # installiert haben, können Sie F # Interactive in der Konsole ausführen, indem Sie `"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"` . Unter Linux oder OS X, ist der Befehl `fsharpi` statt, die entweder in sein sollte `/usr/bin` oder in

`/usr/local/bin` , je nachdem , wie Sie F # installiert - so oder so, sollte der Befehl auf Ihrem sein `PATH` so können Sie `fsharpi` einfach `fsharpi` .

Beispiel für die interaktive Verwendung von F #:

```
> let i = 1 // fsi prompt, declare i
- let j = 2 // declare j
- i+j // compose expression
- ;; // execute commands

val i : int = 1 // fsi output started, this gives the value of i
val j : int = 2 // the value of j
val it : int = 3 // computed expression

> #quit;; //quit fsi
```

Verwenden Sie `#help;;` für Hilfe

Bitte beachten Sie die Verwendung von `;;` um der REPL mitzuteilen, zuvor eingegebene Befehle auszuführen.

Erste Schritte mit F # online lesen: https://riptutorial.com/de/fsharp/topic/817/erste-schritte-mit-f-sharp

Kapitel 2: 1: F # WPF-Code hinter der Anwendung mit FsXaml

Einführung

Die meisten Beispiele, die für die F # WPF-Programmierung gefunden wurden, scheinen sich mit dem MVVM-Muster und einigen wenigen mit MVC zu befassen, aber es gibt so gut wie keine, die richtig zeigt, wie man mit "gutem, altem" Code hinter sich lässt.

Der Code hinter dem Muster ist sehr einfach für das Lehren und Experimentieren zu verwenden. Es wird in zahlreichen Einführungsbüchern und Lernmaterialien im Web verwendet. Deshalb.

In diesen Beispielen wird veranschaulicht, wie ein Code hinter einer Anwendung mit Fenstern, Steuerelementen, Bildern und Symbolen usw. erstellt wird.

Examples

Erstellen Sie einen neuen F # WPF-Code hinter der Anwendung.

Erstellen Sie eine F # -Konsolenanwendung.

Ändern Sie den **Ausgabetyp** der Anwendung in *Windows-Anwendung* .

Fügen Sie das **FsXaml** NuGet-Paket hinzu.

Fügen Sie diese vier Quelldateien in der hier angegebenen Reihenfolge hinzu.

MainWindow.xaml

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="First Demo" Height="200" Width="300">
    <Canvas>
        <Button Name="btnTest" Content="Test" Canvas.Left="10" Canvas.Top="10" Height="28"
Width="72"/>
    </Canvas>
</Window>
```

MainWindow.xaml.fs

```
namespace FirstDemo

type MainWindowXaml = FsXaml.XAML<"MainWindow.xaml">

type MainWindow() as this =
    inherit MainWindowXaml()
```

```
    let whenLoaded _ =
        ()

    let whenClosing _ =
        ()

    let whenClosed _ =
        ()

    let btnTestClick _ =
        this.Title <- "Yup, it works!"

    do
        this.Loaded.Add whenLoaded
        this.Closing.Add whenClosing
        this.Closed.Add whenClosed
        this.btnTest.Click.Add btnTestClick
```

App.xaml

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

App.xaml.fs

```
namespace FirstDemo

open System

type App = FsXaml.XAML<"App.xaml">

module Main =

    [<STAThread; EntryPoint>]
    let main _ =
        let app = App()
        let mainWindow = new MainWindow()
        app.Run(mainWindow) // Returns application's exit code.
```

Löschen Sie die *Program.fs*- Datei aus dem Projekt.

Ändern Sie die **Build-Aktion** in *Ressource* für die beiden XAML-Dateien.

Fügen Sie einen Verweis auf die .NET-Assembly- **UIAutomationTypes hinzu** .

Kompilieren und ausführen

Sie können den Designer nicht zum Hinzufügen von Ereignishandlern verwenden, aber das ist überhaupt kein Problem. Fügen Sie sie einfach manuell in den nachstehenden Code ein, wie Sie es in diesem Beispiel mit den drei Handlern sehen, einschließlich des Handlers für die Test-Schaltfläche.

UPDATE: FsXaml wurde um eine alternative und wahrscheinlich elegantere Methode zum Hinzufügen von Ereignishandlern erweitert. Sie können den Ereignishandler in XAML hinzufügen, genau wie in C #. Sie müssen dies jedoch manuell tun und das entsprechende Member überschreiben, das in Ihrem F # -Typ angezeigt wird. Ich empfehle das.

3: Hinzufügen eines Symbols zu einem Fenster

Es ist eine gute Idee, alle Symbole und Bilder in einem oder mehreren Ordnern aufzubewahren.

Klicken Sie mit der rechten Maustaste auf das Projekt und erstellen Sie mit F # Power Tools / New Folder einen Ordner mit dem Namen Images.

Auf der Festplatte, legen Sie Ihr Symbol in den neuen Ordner *Bilder.*

Klicken Sie in Visual Studio mit der rechten Maustaste auf *Images* , verwenden Sie **Add / Existing Item** , und zeigen Sie *All Files (.*) ** an, um die Symboldatei anzuzeigen, sodass Sie sie auswählen können, und **fügen Sie** sie hinzu.

Wählen Sie die Symboldatei aus, und legen Sie ihre **Build-Aktion** auf *Ressource fest* .

Verwenden Sie in MainWindow.xaml das Icon-Attribut wie folgt. Umliegende Linien werden für den Kontext angezeigt.

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="First Demo" Height="200" Width="300"
    Icon="Images/MainWindow.ico">
<Canvas>
```

Führen Sie vor dem Ausführen eine Neuerstellung durch und nicht nur einen Build. Dies liegt daran, dass Visual Studio die Symboldatei nicht immer in der ausführbaren Datei ablegt, wenn Sie nicht neu erstellen.

Es ist das Fenster und nicht die Anwendung, die jetzt ein Symbol hat. Sie sehen das Symbol zur Laufzeit oben links im Fenster und in der Taskleiste. Der Task-Manager und der Windows-Datei-Explorer zeigen dieses Symbol nicht an, da sie das Anwendungssymbol anstelle des Fenstersymbols anzeigen.

4: Symbol zur Anwendung hinzufügen

Erstellen Sie eine Textdatei mit dem Namen AppIcon.rc mit folgendem Inhalt.

```
1 ICON "AppIcon.ico"
```

Dazu benötigen Sie eine Icon-Datei mit dem Namen AppIcon.ico, aber Sie können die Namen natürlich nach Ihren Wünschen anpassen.

Führen Sie den folgenden Befehl aus.

```
"C:\Program Files (x86)\Windows Kits\10\bin\x64\rc.exe" /v AppIcon.rc
```

Wenn Sie die Datei rc.exe an diesem Ort nicht finden können, suchen Sie sie unter **C: \ Programme (x86) \ Windows Kits** . Wenn Sie es immer noch nicht finden können, laden Sie Windows SDK von Microsoft herunter.

Eine Datei mit dem Namen AppIcon.res wird generiert.

Öffnen Sie in Visual Studio die Projekteigenschaften. Wählen Sie die **Anwendungsseite aus** .

Geben Sie im Textfeld **Ressourcendatei** *AppIcon.res* (oder *Images \ AppIcon.res,* wenn Sie es dort *ablegen*) ein, und schließen Sie die Projekteigenschaften, um sie zu speichern.

Es erscheint eine Fehlermeldung, die besagt "Die eingegebene Ressourcendatei ist nicht vorhanden. Ignorieren Sie diese. Die Fehlermeldung wird nicht erneut angezeigt.

Wiederaufbau Die ausführbare Datei hat dann ein Anwendungssymbol, das im Datei-Explorer angezeigt wird. Während der Ausführung wird dieses Symbol auch im Task-Manager angezeigt.

2: Fügen Sie ein Steuerelement hinzu

Fügen Sie diese beiden Dateien in dieser Reihenfolge über den Dateien für das Hauptfenster hinzu.

MyControl.xaml

```
<UserControl
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d" Height="50" Width="150">
    <Canvas Background="LightGreen">
        <Button Name="btnMyTest" Content="My Test" Canvas.Left="10" Canvas.Top="10"
Height="28" Width="106"/>
    </Canvas>
</UserControl>
```

MyControl.xaml.fs

```
namespace FirstDemo

type MyControlXaml = FsXaml.XAML<"MyControl.xaml">

type MyControl() =
    inherit MyControlXaml()
```

Die **Build-Aktion** für *MyControl.xaml* muss auf *Resource gesetzt sein* .

Natürlich müssen Sie später in der Deklaration von MyControl "as this" hinzufügen, genau wie im Hauptfenster.

Fügen Sie in der Datei **MainWindow.xaml.fs** in der Klasse für MainWindow diese Zeile hinzu

```
let myControl = MyControl()
```

und fügen Sie diese beiden Zeilen im **do-** Bereich der Hauptfensterklasse hinzu.

```
    this.mainCanvas.Children.Add myControl |> ignore
    myControl.btnMyTest.Content <- "We're in business!"
```

Es kann mehr als eine **do-** Sektion in einer Klasse geben, und Sie benötigen sie wahrscheinlich, wenn Sie Code-Behind-Code schreiben.

Das Steuerelement wurde mit einer hellgrünen Hintergrundfarbe versehen, sodass Sie leicht erkennen können, wo es sich befindet.

Beachten Sie, dass das Steuerelement die Schaltfläche des Hauptfensters blockiert. Es liegt nicht im Rahmen dieser Beispiele, Ihnen WPF allgemein beizubringen, daher werden wir das hier nicht beheben.

So fügen Sie Steuerelemente aus Drittanbieter-Bibliotheken hinzu

Wenn Sie Steuerelemente aus Bibliotheken von Drittanbietern in einem C # WPF-Projekt hinzufügen, enthält die XAML-Datei normalerweise solche Zeilen.

```
xmlns:xctk="http://schemas.xceed.com/wpf/xaml/toolkit"
```

Dies funktioniert vielleicht nicht mit FsXaml.

Der Designer und der Compiler akzeptieren diese Zeile, es wird jedoch wahrscheinlich zur Laufzeit eine Ausnahme geben, die besagt, dass der Drittanbieter-Typ beim Lesen der XAML nicht gefunden wurde.

Versuchen Sie stattdessen etwas wie das Folgende.

```
xmlns:xctk="clr-namespace:Xceed.Wpf.Toolkit;assembly=Xceed.Wpf.Toolkit"
```

Dies ist dann ein Beispiel für ein Steuerelement, das von den oben genannten abhängig ist.

```
<xctk:IntegerUpDown Name="tbInput" Increment="1" Maximum="10" Minimum="0" Canvas.Left="13"
Canvas.Top="27" Width="270"/>
```

Die in diesem Beispiel verwendete Bibliothek ist das Extended Wpf Toolkit, das kostenlos über NuGet oder als Installationsprogramm verfügbar ist. Wenn Sie Bibliotheken über NuGet herunterladen, sind die Steuerelemente nicht in der Toolbox verfügbar. Sie werden jedoch im Designer angezeigt, wenn Sie sie manuell in XAML hinzufügen. Die Eigenschaften sind im Bereich Eigenschaften verfügbar.

1: F # WPF-Code hinter der Anwendung mit FsXaml online lesen: https://riptutorial.com/de/fsharp/topic/9008/1--f-sharp-wpf-code-hinter-der-anwendung-mit-fsxaml

Kapitel 3: Aktive Muster

Examples

Einfache aktive Muster

Aktive Muster sind eine spezielle Art der Mustererkennung, in der Sie benannte Kategorien angeben können, in die Ihre Daten fallen können, und diese Kategorien dann in `match` .

So definieren Sie ein aktives Muster, das Zahlen als positiv, negativ oder null klassifiziert:

```
let (|Positive|Negative|Zero|) num =
    if num > 0 then Positive
    elif num < 0 then Negative
    else Zero
```

Dies kann dann in einem Mustervergleichsausdruck verwendet werden:

```
let Sign value =
    match value with
    | Positive -> printf "%d is positive" value
    | Negative -> printf "%d is negative" value
    | Zero -> printf "The value is zero"

Sign -19 // -19 is negative
Sign 2 // 2 is positive
Sign 0 // The value is zero
```

Aktive Muster mit Parametern

Aktive Muster sind nur einfache Funktionen.

Wie Funktionen können Sie zusätzliche Parameter definieren:

```
let (|HasExtension|_|) expected (uri : string) =
    let result = uri.EndsWith (expected, StringComparison.CurrentCultureIgnoreCase)
    match result with
    | true -> Some true
    | _ -> None
```

Dies kann in einem Muster verwendet werden, das auf diese Weise passt:

```
    let isXMLFile uri =
        match uri with
        | HasExtension ".xml" _ -> true
        | _ -> false
```

Aktive Muster können zur Überprüfung und Umwandlung von Funktionsargumenten verwendet werden

Eine interessante, aber eher unbekannte Verwendung von Active Patterns in `F#` besteht darin, dass sie zum Validieren und Transformieren von Funktionsargumenten verwendet werden können.

Betrachten Sie die klassische Art der Argumentvalidierung:

```
// val f : string option -> string option -> string
let f v u =
  let v = defaultArg v "Hello"
  let u = defaultArg u "There"
  v + " " + u

// val g : 'T -> 'T (requires 'T null)
let g v =
  match v with
  | null  -> raise (System.NullReferenceException ())
  | _     -> v.ToString ()
```

Normalerweise fügen wir der Methode Code hinzu, um zu überprüfen, ob die Argumente korrekt sind. Mit Active Patterns in `F#` wir dies verallgemeinern und die Absicht in der Argumentdeklaration deklarieren.

Der folgende Code entspricht dem obigen Code:

```
let inline (|DefaultArg|) dv ov = defaultArg ov dv

let inline (|NotNull|) v =
  match v with
  | null  -> raise (System.NullReferenceException ())
  | _     -> v

// val f : string option -> string option -> string
let f (DefaultArg "Hello" v) (DefaultArg "There" u) = v + " " + u

// val g : 'T -> string (requires 'T null)
let g (NotNull v) = v.ToString ()
```

Für den Benutzer der Funktion `f` und `g` gibt es keinen Unterschied zwischen den beiden verschiedenen Versionen.

```
printfn "%A" <| f (Some "Test") None  // Prints "Test There"
printfn "%A" <| g "Test"              // Prints "Test"
printfn "%A" <| g null                // Will throw
```

Ein Problem ist, wenn Active Patterns Performance-Overhead hinzufügen. Verwenden wir `ILSpy` um `f` und `g` zu dekompilieren, um zu sehen, ob dies der Fall ist.

```
public static string f(FSharpOption<string> _arg2, FSharpOption<string> _arg1)
{
  return Operators.DefaultArg<string>(_arg2, "Hello") + " " +
Operators.DefaultArg<string>(_arg1, "There");
}

public static string g<a>(a _arg1) where a : class
```

```
{
  if (_arg1 != null)
  {
    a a = _arg1;
    return a.ToString();
  }
  throw new NullReferenceException();
}
```

Dank `inline` fügen die Active Patterns keinen zusätzlichen Overhead hinzu, verglichen mit der klassischen Methode der Argumentvalidierung.

Aktive Muster als .NET-API-Wrapper

Active Patterns können verwendet werden, um das Aufrufen einiger .NET-APIs natürlicher zu machen, insbesondere solche, die einen Ausgabeparameter verwenden, um mehr als nur den Funktionsrückgabewert zurückzugeben.

Beispielsweise würden Sie normalerweise die Methode `System.Int32.TryParse` wie folgt `System.Int32.TryParse` :

```
let couldParse, parsedInt = System.Int32.TryParse("1")
if couldParse then printfn "Successfully parsed int: %i" parsedInt
else printfn "Could not parse int"
```

Sie können dies mit dem Pattern-Matching verbessern:

```
match System.Int32.TryParse("1") with
| (true, parsedInt) -> printfn "Successfully parsed int: %i" parsedInt
| (false, _) -> printfn "Could not parse int"
```

Wir können jedoch auch das folgende aktive Muster definieren, das die Funktion `System.Int32.TryParse` :

```
let (|Int|_|) str =
    match System.Int32.TryParse(str) with
    | (true, parsedInt) -> Some parsedInt
    | _ -> None
```

Jetzt können wir folgendes tun:

```
match "1" with
| Int parsedInt -> printfn "Successfully parsed int: %i" parsedInt
| _ -> printfn "Could not parse int"
```

Ein weiterer guter Kandidat für die Umhüllung in aktive Muster sind die APIs für reguläre Ausdrücke:

```
let (|MatchRegex|_|) pattern input =
    let m = System.Text.RegularExpressions.Regex.Match(input, pattern)
    if m.Success then Some m.Groups.[1].Value
```

```
    else None

match "bad" with
| MatchRegex "(good|great)" mood ->
    printfn "Wow, it's a %s day!" mood
| MatchRegex "(bad|terrible)" mood ->
    printfn "Unfortunately, it's a %s day." mood
| _ ->
    printfn "Just a normal day"
```

Vollständige und teilweise aktive Muster

Es gibt zwei Arten von aktiven Mustern, die sich in der Verwendung etwas unterscheiden - vollständig und teilweise.

Vollständige aktive Muster können verwendet werden, wenn Sie alle Ergebnisse auflisten können, z. B. "Ist eine Zahl ungerade oder gerade?"

```
let (|Odd|Even|) number =
  if number % 2 = 0
  then Even
  else Odd
```

Beachten Sie, dass die Definition des aktiven Musters sowohl mögliche Fälle als auch nichts anderes aufführt, und der Hauptteil gibt einen der aufgeführten Fälle zurück. Wenn Sie es in einem Übereinstimmungsausdruck verwenden, ist dies eine vollständige Übereinstimmung:

```
let n = 13
match n with
| Odd -> printf "%i is odd" n
| Even -> printf "%i is even" n
```

Dies ist praktisch, wenn Sie den Eingabebereich in bekannte Kategorien unterteilen möchten, die ihn vollständig abdecken.

Partial Active Patterns hingegen lassen Sie mögliche Ergebnisse explizit ignorieren, indem Sie eine `option` . Ihre Definition verwendet einen Sonderfall von _ für den nicht übereinstimmenden Fall.

```
let (|Integer|_|) str =
  match Int32.TryParse(str) with
  | (true, i) -> Some i
  | _ -> None
```

Auf diese Weise können wir auch dann übereinstimmen, wenn einige Fälle von unserer Parsing-Funktion nicht behandelt werden können.

```
let s = "13"
match s with
| Integer i -> "%i was successfully parsed!" i
| _ -> "%s is not an int" s
```

Teilweise aktive Muster können als eine Form des Testens verwendet werden, ob die Eingabe in eine bestimmte Kategorie im Eingabebereich fällt, während andere Optionen ignoriert werden.

Aktive Muster online lesen: https://riptutorial.com/de/fsharp/topic/962/aktive-muster

Kapitel 4: Aufzeichnungen

Examples

Fügen Sie Mitgliedsfunktionen zu Datensätzen hinzu

```
type person = {Name: string; Age: int} with   // Defines person record
    member this.print() =
        printfn "%s, %i" this.Name this.Age

let user = {Name = "John Doe"; Age = 27} // creates a new person

user.print() // John Doe, 27
```

Grundlegende Verwendung

```
type person = {Name: string; Age: int}   // Defines person record

let user1 = {Name = "John Doe"; Age = 27} // creates a new person
let user2 = {user1 with Age = 28}         // creates a copy, with different Age
let user3 = {user1 with Name = "Jane Doe"; Age = 29} //creates a copy with different Age and
Name

let printUser user =
    printfn "Name: %s, Age: %i" user.Name user.Age

printUser user1 // Name: John Doe, Age: 27
printUser user2 // Name: John Doe, Age: 28
printUser user3 // Name: Jane Doe, Age: 29
```

Aufzeichnungen online lesen: https://riptutorial.com/de/fsharp/topic/1136/aufzeichnungen

Kapitel 5: Der Typ "Einheit"

Examples

Was nützt ein 0-Tupel?

Ein 2-Tupel oder ein 3-Tupel repräsentiert eine Gruppe verwandter Elemente. (Punkte im 2D-Raum, RGB-Werte einer Farbe usw.) Ein 1-Tupel ist nicht sehr nützlich, da es leicht durch ein einzelnes `int` .

Ein 0-Tupel erscheint noch nutzloser, da es absolut *nichts* enthält. Es hat jedoch Eigenschaften, die es in funktionalen Sprachen wie F # sehr nützlich machen. Zum Beispiel hat der 0-Tupel-Typ genau *einen* Wert, normalerweise als `()` . Alle 0-Tupel haben diesen Wert, es handelt sich also im Wesentlichen um einen Singleton-Typ. In den meisten funktionalen Programmiersprachen, einschließlich F #, wird dies die gerufene `unit` Typ.

Funktionen, die `void` in C # zurückgeben, geben den `unit` in F # zurück:

```
let printResult = printfn "Hello"
```

Führen Sie das im interaktiven F # -Interpreter aus, und Sie werden sehen:

```
val printResult : unit = ()
```

Dies bedeutet, dass der Wert `printResult` vom Typ `unit` ist und den Wert `()` (das leere Tupel, der einzige Wert des `unit`).

Funktionen können den `unit` als Parameter verwenden. In F # sehen Funktionen möglicherweise so aus, als würden sie keine Parameter übernehmen. Tatsächlich nehmen sie jedoch einen einzigen Parameter der `unit` . Diese Funktion:

```
let doMath() = 2 + 4
```

ist eigentlich gleichbedeutend mit:

```
let doMath () = 2 + 4
```

Das heißt, eine Funktion, die einen Parameter der `unit` und den `int` Wert 6 zurückgibt. Wenn Sie sich die `int` ansehen, die der interaktive F # -Interpreter bei der Definition dieser Funktion druckt, wird Folgendes angezeigt:

```
val doMath : unit -> int
```

Die Tatsache, dass alle Funktionen mindestens einen Parameter annehmen und einen Wert zurückgeben, auch wenn dieser Wert manchmal ein "nutzloser" Wert wie `()` , bedeutet, dass die

Funktionskomposition in F # viel einfacher ist als in Sprachen, die das nicht haben `unit` - Typ. Aber das ist ein fortgeschritteneres Thema, auf das wir später noch eingehen werden. Denken Sie im Moment daran, dass, wenn Sie `unit` in einer Funktionssignatur oder `()` in den Parametern einer Funktion sehen, dies der 0-Tupel-Typ ist, der dazu dient, zu sagen: "Diese Funktion nimmt oder gibt keine sinnvollen Werte zurück."

Verschiebung der Ausführung von Code

Wir können den `unit` Typ als Funktionsargument verwenden, um Funktionen zu definieren, die erst später ausgeführt werden sollen. Dies ist häufig bei asynchronen Hintergrundaufgaben nützlich, wenn der Haupt-Thread möglicherweise vordefinierte Funktionen des Hintergrund-Threads auslöst, z. B. das Verschieben in eine neue Datei oder wenn eine Let-Bindung nicht sofort ausgeführt werden soll:

```
module Time =
    let now = System.DateTime.Now   // value is set and fixed for duration of program
    let now() = System.DateTime.Now // value is calculated when function is called (each time)
```

Im folgenden Code definieren wir Code zum Starten eines "Arbeiters", der einfach alle 2 Sekunden den Wert ausgibt, an dem er arbeitet. Der Arbeiter gibt dann zwei Funktionen zurück, mit denen er gesteuert werden kann - eine, mit der er zum nächsten Wert bewegt werden kann, und eine, die die Arbeit unterbindet. Dies müssen Funktionen sein, da wir nicht möchten, dass ihre Körper ausgeführt werden, bis wir dies wünschen. Andernfalls würde der Arbeiter sofort zum zweiten Wert wechseln und herunterfahren, ohne etwas getan zu haben.

```
let startWorker value =
    let current = ref value
    let stop = ref false
    let nextValue () = current := !current + 1
    let stopOnNextTick () = stop := true
    let rec loop () = async {
        if !stop then
            printfn "Stopping work."
            return ()
        else
            printfn "Working on %d." !current
            do! Async.Sleep 2000
            return! loop () }
    Async.Start (loop ())
    nextValue, stopOnNextTick
```

Wir können dann einen Arbeiter damit anfangen

```
let nextValue, stopOnNextTick = startWorker 12
```

und die Arbeit beginnt - wenn wir in F # interaktiv sind, werden die Meldungen alle zwei Sekunden in der Konsole ausgedruckt. Wir können dann rennen

```
nextValue ()
```

Die Meldungen zeigen an, dass der Wert, an dem gearbeitet wird, zum nächsten verschoben wurde.

Wenn es Zeit ist, die Arbeit zu beenden, können wir das ausführen

```
stopOnNextTick ()
```

Funktion, die die Abschlussnachricht druckt, und beendet sich dann.

Der `unit` ist hier wichtig, um "keine Eingabe" zu bedeuten - die Funktionen enthalten bereits alle Informationen, die sie zum Arbeiten benötigen, und der Anrufer darf dies nicht ändern.

Der Typ "Einheit" online lesen: https://riptutorial.com/de/fsharp/topic/2513/der-typ--einheit-

Kapitel 6: Designmuster-Implementierung in F

Examples

Datengesteuerte Programmierung in F

Dank Typinferenz und partieller Anwendung in `F#` die datengesteuerte Programmierung kurz und lesbar.

Stellen wir uns vor, wir verkaufen Autoversicherungen. Bevor wir versuchen, es an einen Kunden zu verkaufen, versuchen wir zu ermitteln, ob der Kunde ein gültiger potenzieller Kunde für unser Unternehmen ist, indem er das Geschlecht und das Alter des Kunden überprüft.

Ein einfaches Kundenmodell:

```
type Sex =
  | Male
  | Female

type Customer =
  {
    Name    : string
    Born    : System.DateTime
    Sex     : Sex
  }
```

Als Nächstes möchten wir eine Ausschlussliste (Tabelle) definieren, damit ein Kunde keine Kfz-Versicherung abschließen kann, wenn er mit einer Zeile in der Ausschlussliste übereinstimmt.

```
//  If any row in this list matches the Customer, the customer isn't eligible for the car
insurance.
let exclusionList =
  let __          _   = true
  let olderThan   x y = x < y
  let youngerThan x y = x > y
  [|
//  Description                         Age               Sex
    "Not allowed for senior citizens" , olderThan   65  , __
    "Not allowed for children"        , youngerThan 16  , __
    "Not allowed for young males"     , youngerThan 25  , (=) Male
  |]
```

Aufgrund der Typinferenz und der teilweisen Anwendung ist die Ausschlussliste flexibel und leicht verständlich.

Schließlich definieren wir eine Funktion, die die Ausschlussliste (eine Tabelle) verwendet, um die Kunden in zwei Bereiche aufzuteilen: potenzielle und verweigerte Kunden.

```
// Splits customers into two buckets: potential customers and denied customers.
// The denied customer bucket also includes the reason for denying them the car insurance
let splitCustomers (today : System.DateTime) (cs : Customer []) : Customer
[]*(string*Customer) [] =
  let potential = ResizeArray<_> 16 // ResizeArray is an alias for
System.Collections.Generic.List
  let denied    = ResizeArray<_> 16

  for c in cs do
    let age = today.Year - c.Born.Year
    let sex = c.Sex
    match exclusionList |> Array.tryFind (fun (_, testAge, testSex) -> testAge age && testSex
sex) with
    | Some (description, _, _)  -> denied.Add (description, c)
    | None                      -> potential.Add c

  potential.ToArray (), denied.ToArray ()
```

Zum Abschluss definieren wir einige Kunden und sehen, ob sie potenzielle Kunden für unsere Kfz-Versicherung sind:

```
let customers =
  let c n s y m d: Customer = { Name = n; Born = System.DateTime (y, m, d); Sex = s }
  [|
//    Name                      Sex     Born
    c "Clint Eastwood Jr."      Male    1930 05 31
    c "Bill Gates"              Male    1955 10 28
    c "Melina Gates"            Female  1964 08 15
    c "Justin Drew Bieber"      Male    1994 03 01
    c "Sophie Turner"           Female  1996 02 21
    c "Isaac Hempstead Wright"  Male    1999 04 09
  |]

[<EntryPoint>]
let main argv =
  let potential, denied = splitCustomers (System.DateTime (2014, 06, 01)) customers
  printfn "Potential Customers (%d)\n%A" potential.Length potential
  printfn "Denied Customers (%d)\n%A"    denied.Length    denied
  0
```

Dies druckt:

```
Potential Customers (3)
[|{Name = "Bill Gates";
   Born = 1955-10-28 00:00:00;
   Sex = Male;}; {Name = "Melina Gates";
                  Born = 1964-08-15 00:00:00;
                  Sex = Female;}; {Name = "Sophie Turner";
                                   Born = 1996-02-21 00:00:00;
                                   Sex = Female;}|]
Denied Customers (3)
[|("Not allowed for senior citizens", {Name = "Clint Eastwood Jr.";
                                       Born = 1930-05-31 00:00:00;
                                       Sex = Male;});
  ("Not allowed for young males", {Name = "Justin Drew Bieber";
                                   Born = 1994-03-01 00:00:00;
                                   Sex = Male;});
  ("Not allowed for children", {Name = "Isaac Hempstead Wright";
```

```
                                Born = 1999-04-09 00:00:00;
                                Sex = Male;})|]
```

Designmuster-Implementierung in F # online lesen:
https://riptutorial.com/de/fsharp/topic/3925/designmuster-implementierung-in-f-sharp

Kapitel 7: Diskriminierte Gewerkschaften

Examples

Benennung von Elementen von Tupeln innerhalb diskriminierter Gewerkschaften

Beim Definieren diskriminierter Vereinigungen können Sie Elemente von Tupeltypen benennen und diese Namen während des Mustervergleichs verwenden.

```
type Shape =
    | Circle of diameter:int
    | Rectangle of width:int * height:int

let shapeIsTenWide = function
    | Circle(diameter=10)
    | Rectangle(width=10) -> true
    | _ -> false
```

Durch die Benennung der Elemente diskriminierter Vereinigungen wird die Lesbarkeit des Codes und die Interoperabilität mit C # verbessert. Die angegebenen Namen werden für die Namen der Eigenschaften und die Parameter der Konstruktoren verwendet. Standardmäßig generierte Namen im Interop-Code sind "Item", "Item1", "Item2" ...

Grundlegende diskriminierte Unionsnutzung

Diskriminierte Vereinigungen in F # bieten eine Möglichkeit, Typen zu definieren, die eine beliebige Anzahl unterschiedlicher Datentypen enthalten können. Ihre Funktionalität ähnelt C ++ - Unions oder VB-Varianten, bietet jedoch den zusätzlichen Vorteil, dass sie typensicher ist.

```
// define a discriminated union that can hold either a float or a string
type numOrString =
    | F of float
    | S of string

let str = S "hi" // use the S constructor to create a string
let fl = F 3.5 // use the F constructor to create a float

// you can use pattern matching to deconstruct each type
let whatType x =
    match x with
        | F f -> printfn "%f is a float" f
        | S s -> printfn "%s is a string" s

whatType str // hi is a string
whatType fl // 3.500000 is a float
```

Enum-Style-Vereinigungen

In den Fällen einer diskriminierten Gewerkschaft muss keine Typinformation enthalten sein. Durch

das Weglassen von Typinformationen können Sie eine Vereinigung erstellen, die einfach eine Reihe von Auswahlmöglichkeiten darstellt, ähnlich einer Aufzählung.

```
// This union can represent any one day of the week but none of
// them are tied to a specific underlying F# type
type DayOfWeek = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Konvertierung in und aus Strings mit Reflection

Manchmal ist es notwendig, eine diskriminierte Union in eine und aus einer Zeichenfolge zu konvertieren:

```
module UnionConversion
    open Microsoft.FSharp.Reflection

    let toString (x: 'a) =
        match FSharpValue.GetUnionFields(x, typeof<'a>) with
        | case, _ -> case.Name

    let fromString<'a> (s : string) =
        match FSharpType.GetUnionCases typeof<'a> |> Array.filter (fun case -> case.Name = s)
with
        | [|case|] -> Some(FSharpValue.MakeUnion(case, [||])) :?> 'a)
        | _ -> None
```

Einzelfall diskriminierte Gewerkschaft

Eine einzelne diskriminierte Gewerkschaft ist wie jede andere diskriminierte Gewerkschaft, außer dass es nur einen einzigen Fall gibt.

```
// Define single-case discriminated union type.
type OrderId = OrderId of int
// Construct OrderId type.
let order = OrderId 123
// Deconstruct using pattern matching.
// Parentheses used so compiler doesn't think it is a function definition.
let (OrderId id) = order
```

Es ist nützlich, um die Typsicherheit zu erzwingen und wird häufig in F # verwendet, im Gegensatz zu C # und Java, wo das Erstellen neuer Typen mehr Aufwand verursacht.

Die folgenden zwei alternativen Typdefinitionen führen dazu, dass dieselbe diskriminierte Einheit für einen Fall deklariert wird:

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

Verwendung diskriminierter Ein-Fall-Vereinigungen als Datensätze

Manchmal ist es hilfreich, Unionstypen mit nur einem Fall zu erstellen, um datensatzartige Typen zu implementieren:

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)

let point2 = Point(-2.5, -4.0)
```

Diese werden sehr nützlich, da sie über Pattern Matching auf dieselbe Weise zerlegt werden können wie Tupel-Argumente:

```
let (Point(x1, y1)) = point1
// val x1 : float = 0.0
// val y1 : float = 3.0

let distance (Point(x1,y1)) (Point(x2,y2)) =
    pown (x2-x1) 2 + pown (y2-y1) 2 |> sqrt
// val distance : Point -> Point -> float

distance point1 point2
// val it : float = 7.433034374
```

RequireQualifiedAccess

Mit dem `RequireQualifiedAccess` Attribut müssen Union-Fälle als `MyUnion.MyCase` werden und nicht nur als `MyCase` . Dies verhindert Namenskollisionen im umschließenden Namespace oder Modul.

```
type [<RequireQualifiedAccess>] Requirements =
    None | Single | All

// Uses the DU with qualified access
let noRequirements = Requirements.None

// Here, None still refers to the standard F# option case
let getNothing () = None

// Compiler error unless All has been defined elsewhere
let invalid = All
```

Wenn beispielsweise `System` geöffnet wurde, bezieht sich `Single` auf `System.Single` . Es gibt keine Kollision mit dem Vereinigungsfall `Requirements.Single` .

Rekursive diskriminierte Gewerkschaften

Rekursiver Typ

Diskriminierte Gewerkschaften können rekursiv sein, dh sie können sich in ihrer Definition auf sich selbst beziehen. Das Paradebeispiel hier ist ein Baum:

```
type Tree =
```

```
    | Branch of int * Tree list
    | Leaf of int
```

Als Beispiel definieren wir den folgenden Baum:

```
    1
  2   5
3   4
```

Wir können diesen Baum unter Verwendung unserer rekursiven diskriminierten Vereinigung wie folgt definieren:

```
let leaf1 = Leaf 3
let leaf2 = Leaf 4
let leaf3 = Leaf 5

let branch1 = Branch (2, [leaf1; leaf2])
let tip = Branch (1, [branch1; leaf3])
```

Das Iterieren über den Baum ist nur eine Frage des Musterabgleichs:

```
let rec toList tree =
    match tree with
    | Leaf x -> [x]
    | Branch (x, xs) -> x :: (List.collect toList xs)

let treeAsList = toList tip // [1; 2; 3; 4; 5]
```

Wechselseitig abhängige rekursive Typen

Eine Möglichkeit, eine Rekursion zu erreichen, besteht darin, sich gegenseitig abhängige Typen zu verschachteln.

```
// BAD
type Arithmetic = {left: Expression; op:string; right: Expression}
// illegal because until this point, Expression is undefined
type Expression =
| LiteralExpr of obj
| ArithmeticExpr of Arithmetic
```

Die Definition eines Datensatztyps direkt in einer diskriminierten Union ist veraltet:

```
// BAD
type Expression =
| LiteralExpr of obj
| ArithmeticExpr of {left: Expression; op:string; right: Expression}
// illegal in recent F# versions
```

Sie können die Verwendung `and` der Kette voneinander abhängigen Definitionen Stichwort:

```
// GOOD
type Arithmetic = {left: Expression; op:string; right: Expression}
and Expression =
| LiteralExpr of obj
| ArithmeticExpr of Arithmetic
```

Diskriminierte Gewerkschaften online lesen:
https://riptutorial.com/de/fsharp/topic/1025/diskriminierte-gewerkschaften

Kapitel 8: Einführung in WPF in F

Einführung

In diesem Thema wird veranschaulicht, wie die **funktionale Programmierung** in einer **WPF-Anwendung genutzt wird** . Das erste Beispiel stammt aus einem Beitrag von Māris Krivtežs (siehe Abschnitt " *Anmerkungen* " unten). Es gibt zwei Gründe, dieses Projekt erneut zu besuchen:

1 \ Das Design unterstützt die Trennung von Anliegen, während das Modell rein bleibt und Änderungen funktional weitergegeben werden.

2 \ Die Ähnlichkeit ermöglicht einen einfachen Übergang zur Implementierung von Gjallarhorn.

Bemerkungen

Demo-Projekte für Bibliotheken @GitHub

- FSharp.ViewModule (unter FsXaml)
- Gjallarhorn (Referenzbeispiele)

Māris Krivtežs schrieb zwei großartige Beiträge zu diesem Thema:

- F # XAML-Anwendung - MVVM vs. MVC, bei der die Vor- und Nachteile beider Ansätze hervorgehoben werden.

  Ich denke, keiner dieser XAML-Anwendungsstile profitiert von der funktionalen Programmierung. Ich kann mir vorstellen, dass die ideale Anwendung aus der Ansicht bestehen würde, bei der Ereignisse erzeugt werden und Ereignisse den aktuellen Ansichtszustand haben. Die gesamte Anwendungslogik sollte durch Filtern und Bearbeiten von Ereignissen und Ansichtsmodell behandelt werden. In der Ausgabe sollte ein neues Ansichtsmodell erstellt werden, das an die Ansicht gebunden ist.

- F # XAML - ereignisgesteuerte MVVM wie in dem obigen Thema überarbeitet.

Examples

FSharp.ViewModule

Unsere Demo-App besteht aus einer Anzeigetafel. Das Score-Modell ist ein unveränderlicher Datensatz. Die Ereignisse der Anzeigetafel sind in einem Unionstyp enthalten.

```
namespace Score.Model

type Score = { ScoreA: int ; ScoreB: int }
type ScoringEvent = IncA | DecA | IncB | DecB | NewGame
```

Änderungen werden durch Abhören von Ereignissen verbreitet und das Ansichtsmodell entsprechend aktualisiert. Anstatt wie bei OOP Mitglieder zum Modelltyp hinzuzufügen, deklarieren wir ein separates Modul zum Hosten der zulässigen Operationen.

```
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Score =
    let zero = {ScoreA = 0; ScoreB = 0}
    let update score event =
        match event with
        | IncA -> {score with ScoreA = score.ScoreA + 1}
        | DecA -> {score with ScoreA = max (score.ScoreA - 1) 0}
        | IncB -> {score with ScoreB = score.ScoreB + 1}
        | DecB -> {score with ScoreB = max (score.ScoreB - 1) 0}
        | NewGame -> zero
```

Unser Ansichtsmodell leitet sich von `EventViewModelBase<'a>` , die über die Eigenschaft `EventStream` vom Typ `IObservable<'a>` . In diesem Fall haben die Ereignisse, die wir abonnieren möchten, den Typ `ScoringEvent` .

Der Controller behandelt die Ereignisse funktional. Die Signatur `Score -> ScoringEvent -> Score` zeigt uns, dass der aktuelle Wert des Modells bei Auftreten eines Ereignisses in einen neuen Wert umgewandelt wird. Dies ermöglicht, dass unser Modell rein bleibt, unser Sichtmodell jedoch nicht.

Ein `eventHandler` ist für die `eventHandler` des Ansichtsstatus verantwortlich. Von `EventViewModelBase<'a>` wir `EventValueCommand` und `EventValueCommandChecked` , um die Ereignisse mit den Befehlen zu `EventValueCommandChecked` .

```
namespace Score.ViewModel

open Score.Model
open FSharp.ViewModule

type MainViewModel(controller : Score -> ScoringEvent -> Score) as self =
    inherit EventViewModelBase<ScoringEvent>()

    let score = self.Factory.Backing(<@ self.Score @>, Score.zero)

    let eventHandler ev = score.Value <- controller score.Value ev

    do
        self.EventStream
        |> Observable.add eventHandler

    member this.IncA = this.Factory.EventValueCommand(IncA)
    member this.DecA = this.Factory.EventValueCommandChecked(DecA, (fun _ -> this.Score.ScoreA
> 0), [ <@@ this.Score @@> ])
    member this.IncB = this.Factory.EventValueCommand(IncB)
    member this.DecB = this.Factory.EventValueCommandChecked(DecB, (fun _ -> this.Score.ScoreB
> 0), [ <@@ this.Score @@> ])
    member this.NewGame = this.Factory.EventValueCommand(NewGame)

    member __.Score = score.Value
```

Im Code hinter der Datei (* .xaml.fs) wird alles zusammengesetzt, dh die Aktualisierungsfunktion (`controller`) wird in das `MainViewModel` .

```
namespace Score.Views

open FsXaml

type MainView = XAML<"MainWindow.xaml">

type CompositionRoot() =
    member __.ViewModel = Score.ViewModel.MainViewModel(Score.Model.Score.update)
```

Der Typ `CompositionRoot` dient als Wrapper, auf den in der XAML-Datei verwiesen wird.

```
<Window.Resources>
    <ResourceDictionary>
        <local:CompositionRoot x:Key="CompositionRoot"/>
    </ResourceDictionary>
</Window.Resources>
<Window.DataContext>
    <Binding Source="{StaticResource CompositionRoot}" Path="ViewModel" />
</Window.DataContext>
```

Ich werde nicht tiefer in die XAML-Datei eintauchen, da es grundlegende WPF-Sachen ist. Das gesamte Projekt ist auf GitHub zu finden.

Gjallarhorn

Die `IObservable<'a>` in der Gjallarhorn-Bibliothek implementieren `IObservable<'a>` , wodurch die Implementierung bekannt wird (erinnern Sie sich an die `EventStream` Eigenschaft aus dem FSharp.ViewModule-Beispiel). Die einzige wirkliche Änderung an unserem Modell ist die Reihenfolge der Argumente der Aktualisierungsfunktion. Außerdem verwenden wir jetzt den Begriff *Nachricht* anstelle von *Ereignis* .

```
namespace ScoreLogic.Model

type Score = { ScoreA: int ; ScoreB: int }
type ScoreMessage = IncA | DecA | IncB | DecB | NewGame

// Module showing allowed operations
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Score =
    let zero = {ScoreA = 0; ScoreB = 0}
    let update msg score =
        match msg with
        | IncA -> {score with ScoreA = score.ScoreA + 1}
        | DecA -> {score with ScoreA = max (score.ScoreA - 1) 0}
        | IncB -> {score with ScoreB = score.ScoreB + 1}
        | DecB -> {score with ScoreB = max (score.ScoreB - 1) 0}
        | NewGame -> zero
```

Um eine Benutzeroberfläche mit Gjallarhorn zu erstellen, erstellen wir anstelle von Klassen zur Unterstützung der Datenbindung einfache Funktionen, die als `Component` . In ihrem Konstruktor die erste Argument `source` ist vom Typ `BindingSource` (definiert in Gjallarhorn.Bindable) und verwendet , um das Modell zu der Ansicht auf der Karte, und Ereignisse aus der Sicht zurück in Nachrichten.

```
namespace ScoreLogic.Model

open Gjallarhorn
open Gjallarhorn.Bindable

module Program =

    // Create binding for entire application.
    let scoreComponent source (model : ISignal<Score>) =
        // Bind the score to the view
        model |> Binding.toView source "Score"

        [
            // Create commands that turn into ScoreMessages
            source |> Binding.createMessage "NewGame" NewGame
            source |> Binding.createMessage "IncA" IncA
            source |> Binding.createMessage "DecA" DecA
            source |> Binding.createMessage "IncB" IncB
            source |> Binding.createMessage "DecB" DecB
        ]
```

Die aktuelle Implementierung unterscheidet sich von der Version von FSharp.ViewModule darin, dass CanExecute für zwei Befehle noch nicht ordnungsgemäß implementiert ist. Listet auch die Installation der Anwendung auf.

```
namespace ScoreLogic.Model

open Gjallarhorn
open Gjallarhorn.Bindable

module Program =

    // Create binding for entire application.
    let scoreComponent source (model : ISignal<Score>) =
        let aScored = Mutable.create false
        let bScored = Mutable.create false

        // Bind the score itself to the view
        model |> Binding.toView source "Score"

        // Subscribe to changes of the score
        model |> Signal.Subscription.create
            (fun currentValue ->
                aScored.Value <- currentValue.ScoreA > 0
                bScored.Value <- currentValue.ScoreB > 0)
            |> ignore

        [
            // Create commands that turn into ScoreMessages
            source |> Binding.createMessage "NewGame" NewGame
            source |> Binding.createMessage "IncA" IncA
            source |> Binding.createMessageChecked "DecA" aScored DecA
            source |> Binding.createMessage "IncB" IncB
            source |> Binding.createMessageChecked "DecB" bScored DecB
        ]

    let application =
        // Create our score, wrapped in a mutable with an atomic update function
        let score = new AsyncMutable<_>(Score.zero)
```

```
        // Create our 3 functions for the application framework

        // Start with the function to create our model (as an ISignal<'a>)
        let createModel () : ISignal<_> = score :> _

        // Create a function that updates our state given a message
        // Note that we're just taking the message, passing it directly to our model's update
function,
        // then using that to update our core "Mutable" type.
        let update (msg : ScoreMessage) : unit = Score.update msg |> score.Update |> ignore

        // An init function that occurs once everything's created, but before it starts
        let init () : unit = ()

        // Start our application
        Framework.application createModel init update scoreComponent
```

`MainWindow` Einrichtung der entkoppelten Ansicht, wobei der `MainWindow` Typ und die logische Anwendung kombiniert werden.

```
namespace ScoreBoard.Views

open System

open FsXaml
open ScoreLogic.Model

// Create our Window
type MainWindow = XAML<"MainWindow.xaml">

module Main =
    [<STAThread>]
    [<EntryPoint>]
    let main _ =
        // Run using the WPF wrappers around the basic application framework
        Gjallarhorn.Wpf.Framework.runApplication System.Windows.Application MainWindow
Program.application
```

Dies fasst die Kernkonzepte zusammen. Weitere Informationen und ein ausführlicheres Beispiel finden Sie in Reed Copseys Beitrag . Das *Christmas Trees*- Projekt zeigt einige Vorteile dieses Ansatzes auf:

- Wir müssen das Modell effektiv (manuell) in eine benutzerdefinierte Sammlung von Ansichtsmodellen kopieren, verwalten und das Modell manuell aus den Ergebnissen erstellen.
- Aktualisierungen innerhalb von Sammlungen werden auf transparente Weise durchgeführt, wobei ein reines Modell erhalten bleibt.
- Die Logik und Sichtweise werden von zwei verschiedenen Projekten gehostet, die die Trennung von Anliegen hervorheben.

Einführung in WPF in F # online lesen: https://riptutorial.com/de/fsharp/topic/8758/einfuhrung-in-wpf-in-f-sharp

Kapitel 9: F # auf .NET Core

Examples

Ein neues Projekt über dotnet CLI erstellen

Nachdem Sie die .NET CLI-Tools installiert haben, können Sie mit dem folgenden Befehl ein neues Projekt erstellen:

```
dotnet new --lang f#
```

Dadurch wird ein Befehlszeilenprogramm erstellt.

Erster Projektablauf

Erstellen Sie ein neues Projekt

```
dotnet new -l f#
```

Stellen Sie alle in project.json aufgelisteten Pakete wieder her

```
dotnet restore
```

Eine project.lock.json-Datei sollte geschrieben werden.

Führen Sie das Programm aus

```
dotnet run
```

Das obige wird den Code bei Bedarf kompilieren.

Die Ausgabe des durch `dotnet new -lf#` erstellten Standardprojekts enthält Folgendes:

```
Hello World!
[|||]
```

F # auf .NET Core online lesen: https://riptutorial.com/de/fsharp/topic/4404/f-sharp-auf--net-core

Kapitel 10: F # Tipps und Tricks zur Leistung

Examples

Verwenden der Schwanzrekursion für eine effiziente Iteration

Viele Entwickler fragen sich, wie man eine `for-loop` schreibt, die früh `break` , da `F#` keine `break` , `continue` oder `return` . Die Antwort in `F#` ist die Verwendung der Tail-Rekursion, die eine flexible und idiomatische Methode zur Iteration ist und gleichzeitig eine hervorragende Leistung bietet.

`tryFind` wir möchten `tryFind` for `List` implementieren. Wenn `F#` die `return` unterstützt, würden wir `tryFind` so schreiben:

```
let tryFind predicate vs =
  for v in vs do
    if predicate v then
      return Some v
  None
```

Das funktioniert in `F#` . Stattdessen schreiben wir die Funktion mit Tail-Rekursion:

```
let tryFind predicate vs =
  let rec loop = function
    | v::vs -> if predicate v then
                   Some v
               else
                   loop vs
    | _ -> None
  loop vs
```

Tail-Rekursion ist in `F#` performant. Wenn der `F#` -Compiler erkennt, dass eine Funktion Tail-Recursive ist, schreibt sie die Rekursion in eine effiziente `while-loop` . Mit `ILSpy` können wir sehen , dass dies für unsere Funktion wahr `loop` :

```
internal static FSharpOption<a> loop@3-10<a>(FSharpFunc<a, bool> predicate, FSharpList<a>
_arg1)
{
  while (_arg1.TailOrNull != null)
  {
    FSharpList<a> fSharpList = _arg1;
    FSharpList<a> vs = fSharpList.TailOrNull;
    a v = fSharpList.HeadOrDefault;
    if (predicate.Invoke(v))
    {
      return FSharpOption<a>.Some(v);
    }
    FSharpFunc<a, bool> arg_2D_0 = predicate;
    _arg1 = vs;
    predicate = arg_2D_0;
  }
  return null;
}
```

Abgesehen von einigen unnötigen Zuweisungen (wodurch der JIT-er hoffentlich beseitigt) ist dies im Wesentlichen eine effiziente Schleife.

Darüber hinaus ist die Rekursion des Schwanzes für `F#` idiomatisch, da wir den veränderlichen Zustand vermeiden können. Betrachten Sie eine `sum` , die alle Elemente in einer `List` summiert. Ein offensichtlicher erster Versuch wäre folgender:

```
let sum vs =
  let mutable s = LanguagePrimitives.GenericZero
  for v in vs do
    s <- s + v
  s
```

Wenn wir die Schleife in Tail-Rekursion umschreiben, können wir den veränderbaren Zustand vermeiden:

```
let sum vs =
  let rec loop s = function
    | v::vs -> loop (s + v) vs
    | _ -> s
  loop LanguagePrimitives.GenericZero vs
```

Aus Effizienzgründen wandelt der `F#` -Compiler dies in eine `while-loop` , die den veränderbaren Zustand verwendet.

Messen und überprüfen Sie Ihre Leistungsannahmen

Dieses Beispiel wurde mit Blick auf `F#` geschrieben, aber die Ideen sind in allen Umgebungen anwendbar

Die erste Regel bei der Optimierung der Leistung besteht darin, sich nicht auf die Annahme zu verlassen. Messen und überprüfen Sie immer Ihre Annahmen.

Da wir nicht direkt Maschinencode schreiben, lässt sich schwer vorhersagen, wie der Compiler und JIT: er Ihr Programm in Maschinencode umwandeln. Aus diesem Grund müssen wir die Ausführungszeit messen, um zu sehen, dass wir die erwartete Leistungsverbesserung erzielen, und sicherstellen, dass das eigentliche Programm keinen versteckten Aufwand enthält.

Bei der Überprüfung handelt es sich um einen recht einfachen Prozess, bei dem die ausführbare Datei mithilfe von Tools wie ILSpy zurückentwickelt wird . Der JIT: er macht Verification dadurch komplizierter, dass das Anzeigen des tatsächlichen Maschinencodes schwierig ist, aber machbar ist. Die Untersuchung des `IL-code` führt jedoch meist zu großen Vorteilen.

Das schwierigere Problem ist Messen; schwieriger, weil es schwierig ist, realistische Situationen aufzubauen, in denen Verbesserungen des Codes gemessen werden können. Messen ist immer noch von unschätzbarem Wert.

Analyse einfacher F # -Funktionen

`1..n` wir uns einige einfache `F#` -Funktionen an, die alle ganzzahligen Zahlen in `1..n` auf

verschiedene Weise ansammeln. Da es sich bei dem Bereich um eine einfache Arithmetikreihe handelt, kann das Ergebnis direkt berechnet werden. In diesem Beispiel durchlaufen wir jedoch den Bereich.

Zuerst definieren wir einige nützliche Funktionen zum Messen der Zeit, die eine Funktion benötigt:

```
// now () returns current time in milliseconds since start
let now : unit -> int64 =
  let sw = System.Diagnostics.Stopwatch ()
  sw.Start ()
  fun () -> sw.ElapsedMilliseconds

// time estimates the time 'action' repeated a number of times
let time repeat action : int64*'T =
  let v = action ()  // Warm-up and compute value

  let b = now ()
  for i = 1 to repeat do
    action () |> ignore
  let e = now ()

  e - b, v
```

`time` läuft eine Aktion wiederholt ab. Wir müssen die Tests einige hundert Millisekunden lang ausführen, um die Varianz zu reduzieren.

Dann definieren wir einige Funktionen, die alle ganzen Zahlen in `1..n` auf unterschiedliche Weise ansammeln.

```
// Accumulates all integers 1..n using 'List'
let accumulateUsingList n =
  List.init (n + 1) id
  |> List.sum

// Accumulates all integers 1..n using 'Seq'
let accumulateUsingSeq n =
  Seq.init (n + 1) id
  |> Seq.sum

// Accumulates all integers 1..n using 'for-expression'
let accumulateUsingFor n =
  let mutable sum = 0
  for i = 1 to n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over list range
let accumulateUsingForEachOverList n =
  let mutable sum = 0
  for i in [1..n] do
```

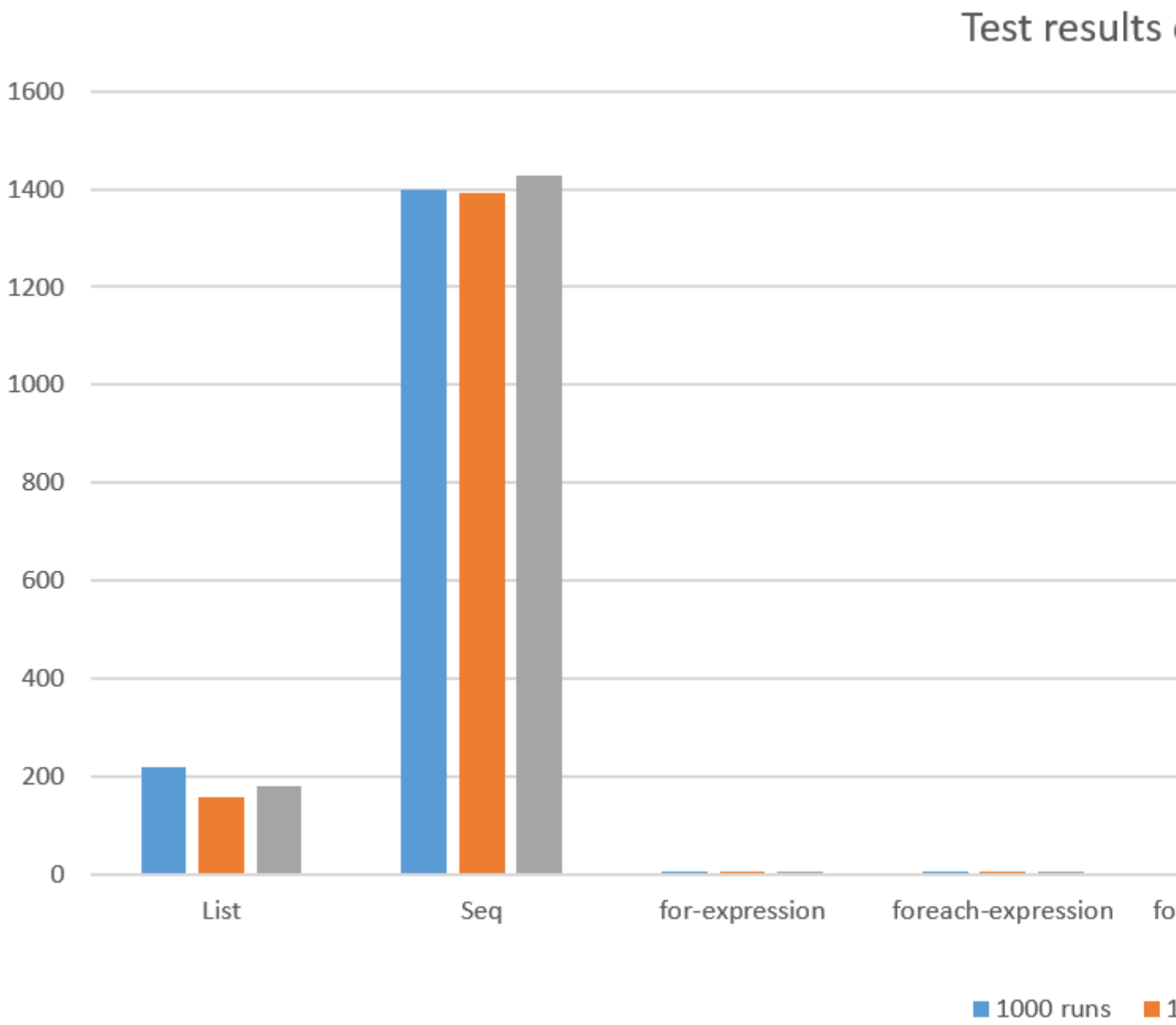
```
    sum <- sum + i
  sum

// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum

// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int

// Accumulates all integers n..1 using 'for-expression' in reverse order
let accumulateUsingReverseFor n =
  let mutable sum = 0
  for i = n downto 1 do
    sum <- sum + i
  sum

// Accumulates all 64 integers n..1 using 'tail-recursion' in reverse order
let accumulateUsingReverseTailRecursion n =
  let rec loop sum i =
    if i > 0 then
      loop (sum + i) (i - 1)
    else
      sum
  loop 0 n
```

Wir gehen davon aus, dass das Ergebnis dasselbe ist (mit Ausnahme einer der Funktionen, die ein Inkrement von `2`), aber es besteht ein Leistungsunterschied. Um dies zu messen, ist folgende Funktion definiert:

```
let testRun (path : string) =
  use testResult = new System.IO.StreamWriter (path)
  let write   (l : string)  = testResult.WriteLine l
  let writef  fmt           = FSharp.Core.Printf.kprintf write fmt

  write "Name\tTotal\tOuter\tInner\tCC0\tCC1\tCC2\tTime\tResult"

  // total is the total number of iterations being executed
  let total   = 10000000
  // outers let us variate the relation between the inner and outer loop
  //  this is often useful when the algorithm allocates different amount of memory
  //  depending on the input size. This can affect cache locality
  let outers  = [| 1000; 10000; 100000 |]
  for outer in outers do
    let inner = total / outer

    // multiplier is used to increase resolution of certain tests that are significantly
    //  faster than the slower ones

    let testCases =
      [|
    //   Name of test                         multiplier    action
```

```
        "List"                              , 1           , accumulateUsingList
        "Seq"                               , 1           , accumulateUsingSeq
        "for-expression"                    , 100         , accumulateUsingFor
        "foreach-expression"                , 100         , accumulateUsingForEach
        "foreach-expression over List"      , 1           , accumulateUsingForEachOverList
        "foreach-expression increment of 2" , 1           , accumulateUsingForEachStep2
        "foreach-expression over 64 bit"    , 1           , accumulateUsingForEach64
        "reverse for-expression"            , 100         , accumulateUsingReverseFor
        "reverse tail-recursion"            , 100         ,
accumulateUsingReverseTailRecursion
      |]
    for name, multiplier, a in testCases do
      System.GC.Collect (2, System.GCCollectionMode.Forced, true)
      let cc g = System.GC.CollectionCount g

      printfn "Accumulate using %s with outer=%d and inner=%d ..." name outer inner

      // Collect collection counters before test run
      let pcc0, pcc1, pcc2 = cc 0, cc 1, cc 2

      let ms, result       = time (outer*multiplier) (fun () -> a inner)
      let ms               = (float ms / float multiplier)

      // Collect collection counters after test run
      let acc0, acc1, acc2 = cc 0, cc 1, cc 2
      let cc0, cc1, cc2    = acc0 - pcc0, acc1 - pcc1, acc1 - pcc1
      printfn "  ... took: %f ms, GC collection count %d,%d,%d and produced %A" ms cc0 cc1 cc2
result

      writef "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%f\t%d" name total outer inner cc0 cc1 cc2 ms result
```

Das Testergebnis beim Ausführen unter .NET 4.5.2 x64:



Wir sehen dramatische Unterschiede und einige der Ergebnisse sind unerwartet schlecht.

Schauen wir uns die schlechten Fälle an:

Liste

```
// Accumulates all integers 1..n using 'List'
let accumulateUsingList n =
  List.init (n + 1) id
  |> List.sum
```

Was hier passiert, ist eine vollständige Liste mit allen ganzen Zahlen `1..n` die mit einer Summe erstellt und reduziert wird. Dies sollte teurer sein als nur das Iterieren und Akkumulieren über den Bereich, es scheint etwa ~ 42x langsamer als die for-Schleife zu sein.

Darüber hinaus sehen wir, dass der GC während des Testlaufs etwa 100x ausgeführt wurde, da

der Code viele Objekte zugewiesen hat. Dies kostet auch CPU.

Seq

```
// Accumulates all integers 1..n using 'Seq'
let accumulateUsingSeq n =
  Seq.init (n + 1) id
  |> Seq.sum
```

Die `Seq` Version weist keine vollständige `List` daher ist es etwas überraschend, dass diese ~ 270x langsamer ist als die for-Schleife. Außerdem sehen wir, dass der GC 661x ausgeführt hat.

`Seq` ist ineffizient, wenn der Arbeitsaufwand pro Artikel sehr gering ist (in diesem Fall werden zwei ganze Zahlen zusammengefasst).

Es geht darum, niemals `Seq` . Der Punkt ist zu messen.

(**manofstick edit:** `Seq.init` ist der Täter dieses schwerwiegenden Leistungsproblems. Es ist viel effizienter, anstelle von `Seq.init (n+1) id` den Ausdruck `{ 0 .. n }` verwenden. Dies wird noch viel effizienter Wenn dieses PR zusammengeführt und veröffentlicht wird, `Seq.init ... |> Seq.sum` die ursprüngliche `Seq.init ... |> Seq.sum` auch nach der Veröffentlichung immer noch langsam, aber etwas kontrapunktisch, `Seq.init ... |> Seq.map id |> Seq.sum` wird ziemlich schnell sein, um die Abwärtskompatibilität mit der Implementierung von `Seq.init` aufrechtzuerhalten, die `Current` anfangs berechnet, sondern sie in ein `Lazy` Objekt `Seq.init` - obwohl auch dies aufgrund von etwas besser sein sollte Diese PR . Hinweis für den Redakteur: Entschuldigung, das ist eine Art wandernde Notizen, aber ich möchte nicht, dass die Leute von Seq abgeschoben werden, wenn die Verbesserung unmittelbar *bevorsteht. Wenn dies der Fall ist, wäre es gut, die Charts zu aktualisieren das sind auf dieser Seite.*)

foreach-expression über List

```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over list range
let accumulateUsingForEachOverList n =
  let mutable sum = 0
  for i in [1..n] do
    sum <- sum + i
  sum
```

Der Unterschied zwischen diesen beiden Funktionen ist sehr subtil, der Leistungsunterschied beträgt jedoch nicht ~ 76x. Warum? Lassen Sie uns den fehlerhaften Code zurückentwickeln:

```
public static int accumulateUsingForEach(int n)
{
  int sum = 0;
  int i = 1;
```

```
  if (n >= i)
  {
    do
    {
      sum += i;
      i++;
    }
    while (i != n + 1);
  }
  return sum;
}

public static int accumulateUsingForEachOverList(int n)
{
  int sum = 0;
  FSharpList<int> fSharpList =
SeqModule.ToList<int>(Operators.CreateSequence<int>(Operators.OperatorIntrinsics.RangeInt32(1,
1, n)));
  for (FSharpList<int> tailOrNull = fSharpList.TailOrNull; tailOrNull != null; tailOrNull =
fSharpList.TailOrNull)
  {
    int i = fSharpList.HeadOrDefault;
    sum += i;
    fSharpList = tailOrNull;
  }
  return sum;
}
```

`accumulateUsingForEach` wird als effiziente `while` Schleife implementiert, aber `for i in [1..n]` wird `for i in [1..n]` konvertiert:

```
FSharpList<int> fSharpList =
  SeqModule.ToList<int>(
    Operators.CreateSequence<int>(
      Operators.OperatorIntrinsics.RangeInt32(1, 1, n)));
```

Dies bedeutet, dass wir zuerst eine `Seq` über `1..n` und schließlich `ToList` .

Teuer.

foreach-expression-Inkrement von 2

```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum
```

Wieder ist der Unterschied zwischen diesen beiden Funktionen geringfügig, aber der

Leistungsunterschied ist brutal: ~ 25x

Lassen Sie uns noch einmal `ILSpy` ausführen:

```
public static int accumulateUsingForEachStep2(int n)
{
  int sum = 0;
  IEnumerable<int> enumerable = Operators.OperatorIntrinsics.RangeInt32(1, 2, n);
  foreach (int i in enumerable)
  {
    sum += i;
  }
  return sum;
}
```

Ein `Seq` wird über `1..2..n` und dann mit dem Enumerator über `Seq` iteriert.

Wir erwarteten, dass `F#` so etwas kreiert:

```
public static int accumulateUsingForEachStep2(int n)
{
  int sum = 0;
  for (int i = 1; i < n; i += 2)
  {
    sum += i;
  }
  return sum;
}
```

`F#` -Compiler unterstützt jedoch nur effizient für Schleifen über int32-Bereiche, die um eins erhöht werden. In allen anderen Fällen wird auf `Operators.OperatorIntrinsics.RangeInt32` . Was das nächste überraschende Ergebnis erklären wird

foreach-expression über 64 bit

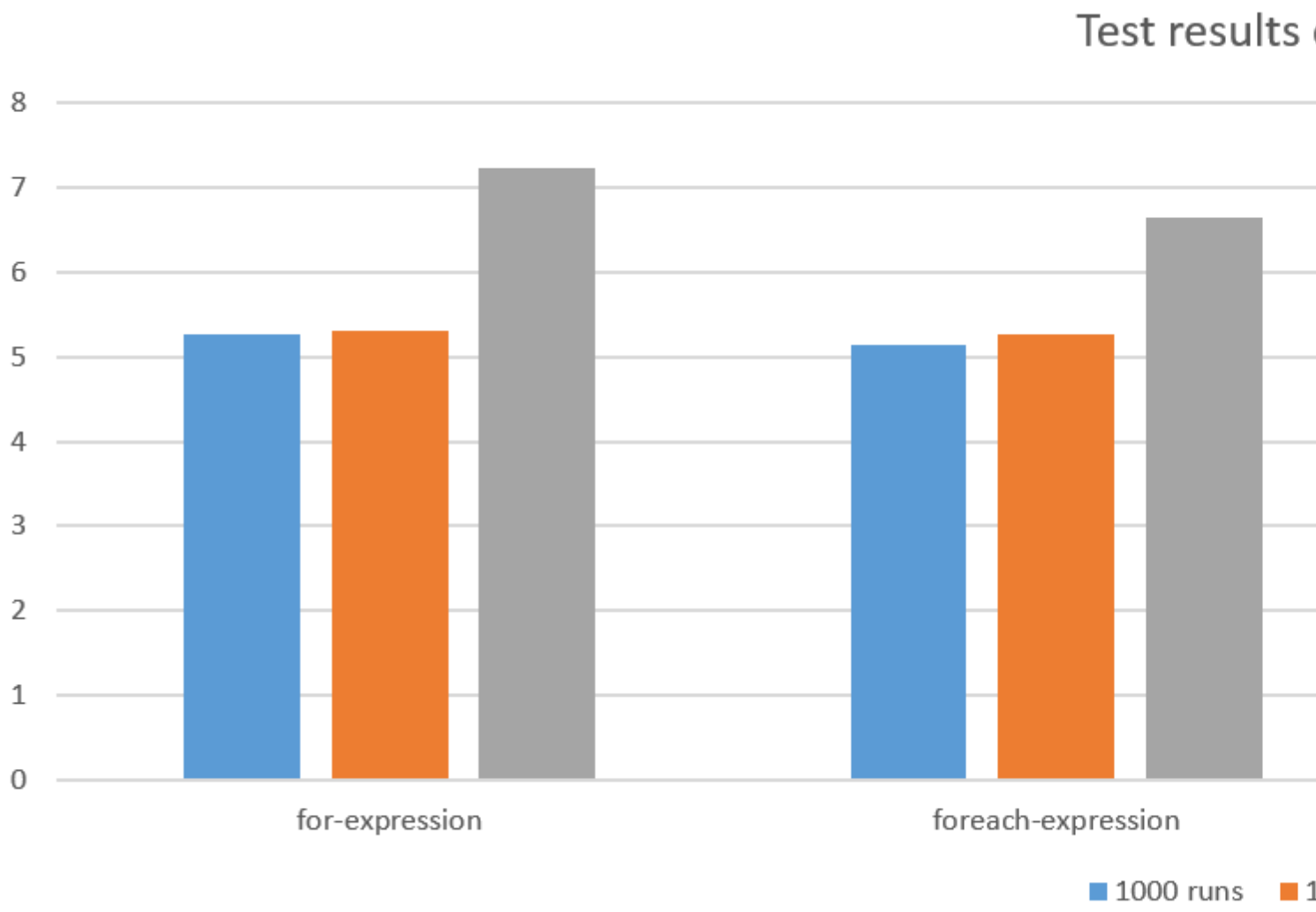
```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int
```

Dies ist ~ 47x langsamer als die for-Schleife. Der einzige Unterschied besteht darin, dass wir über 64-Bit-Ganzzahlen iterieren. `ILSpy` zeigt uns warum:

```
public static int accumulateUsingForEach64(int n)
{
  long sum = 0L;
  IEnumerable<long> enumerable = Operators.OperatorIntrinsics.RangeInt64(1L, 1L, (long)n);
  foreach (long i in enumerable)
  {
    sum += i;
  }
  return (int)sum;
}
```

`F#` unterstützt nur effizient für Schleifen für `int32` Zahlen die Fallback-`Operators.OperatorIntrinsics.RangeInt64` .

Die anderen Fälle verhalten sich ungefähr ähnlich:



Der Grund, warum sich die Leistung bei größeren Testläufen verschlechtert, ist, dass der Aufwand für das Aufrufen der `action` wächst, da wir immer weniger Arbeit in `action` .

Eine Schleife auf `0` kann manchmal zu Leistungsvorteilen führen, da dadurch ein CPU-Register gespeichert werden könnte. In diesem Fall hat die CPU jedoch Register, um zu sparen, sodass es keinen Unterschied zu machen scheint.

Fazit

Das Messen ist wichtig, weil wir sonst denken könnten, dass alle diese Alternativen gleichwertig sind, aber einige Alternativen sind ~ 270x langsamer als andere.

Der Überprüfungsschritt beinhaltet das Reverse Engineering der ausführbaren Datei. Dies hilft uns zu erklären, *warum* wir die erwartete Leistung erzielt haben oder nicht. Darüber hinaus hilft uns die Verifizierung, die Leistung vorherzusagen, wenn es zu schwierig ist, eine korrekte Messung durchzuführen.

Es ist schwierig, die Leistung dort immer vorherzusagen. Messen, immer Überprüfen Sie Ihre Leistungsannahmen.

Vergleich verschiedener F # -Datenpipelines

In `F#` gibt es viele Optionen zum Erstellen von Datenpipelines, z. B. `List` , `Seq` und `Array` .

Welche Daten-Pipeline ist aus Sicht des Speichers und der Leistung vorzuziehen?

Um dies zu beantworten, vergleichen wir Leistung und Speicherauslastung mit verschiedenen Pipelines.

Datenpipeline

Um den Overhead zu messen, verwenden wir eine Datenpipeline mit niedrigen CPU-Kosten pro verarbeitetem Artikel:

```
let seqTest n =
  Seq.init (n + 1) id
  |> Seq.map    int64
  |> Seq.filter (fun v -> v % 2L = 0L)
  |> Seq.map    ((+) 1L)
  |> Seq.sum
```

Wir werden gleichwertige Pipelines für alle Alternativen erstellen und vergleichen.

Wir werden die Größe von `n` variieren, aber die Gesamtzahl der Arbeit sei gleich.

Alternativen für die Datenpipeline

Wir werden die folgenden Alternativen vergleichen:

1. Imperativer Code
2. Array (nicht faul)
3. Liste (nicht faul)
4. LINQ (Lazy Pull Stream)
5. Seq (Lazy Pull Stream)
6. Nessos (Lazy Pull / Push Stream)
7. PullStream (Vereinfachter Pullstream)
8. PushStream (Simplistic Push Stream)

Obwohl es sich nicht um eine Datenpipeline handelt, werden wir sie mit `Imperative` Code vergleichen, da diese am ehesten der Art entspricht, wie die CPU Code ausführt. Dies sollte der schnellste Weg sein, um das Ergebnis zu berechnen, mit dem wir den Performance-Overhead von Datenpipelines messen können.

`Array` und `List` berechnen in jedem Schritt ein vollständiges `Array` / eine `List` daher erwarten wir Speicherüberhang.

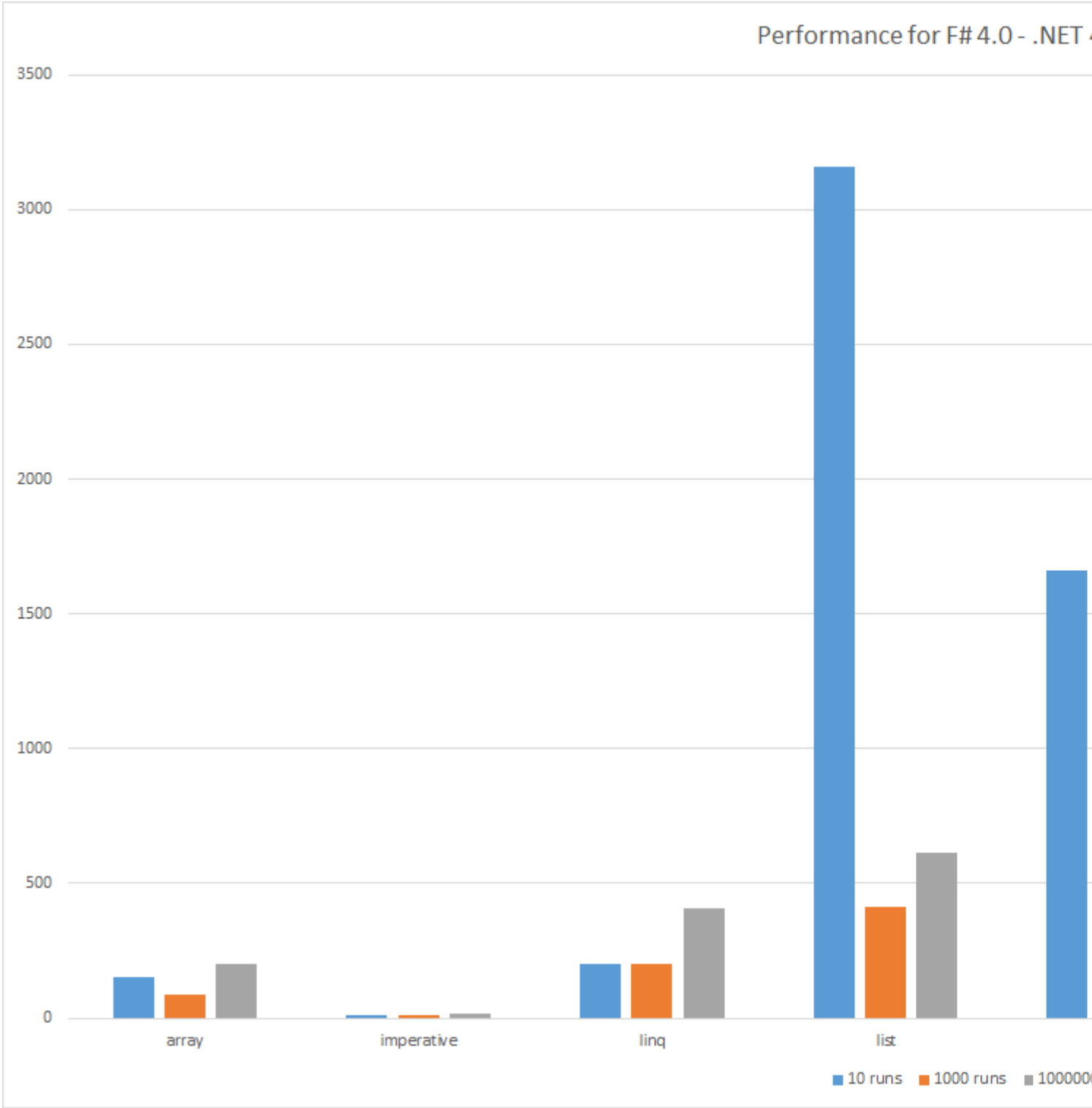
`LINQ` und `Seq` basieren beide auf `IEnumerable<'T>` was ein Lazy-Pull-Stream ist (Pull bedeutet, dass

der Consumer-Stream Daten aus dem Producer-Stream zieht). Wir erwarten daher, dass Leistung und Speicherbedarf identisch sind.

`Nessos` ist eine leistungsstarke Stream-Bibliothek, die Push & Pull (wie Java `Stream`) unterstützt.

PullStream und PushStream sind vereinfachte Implementierungen von `Pull` & `Push` Streams.

Leistung Ergebnisse beim Ausführen auf: F # 4.0 - .NET 4.6.1 - x64



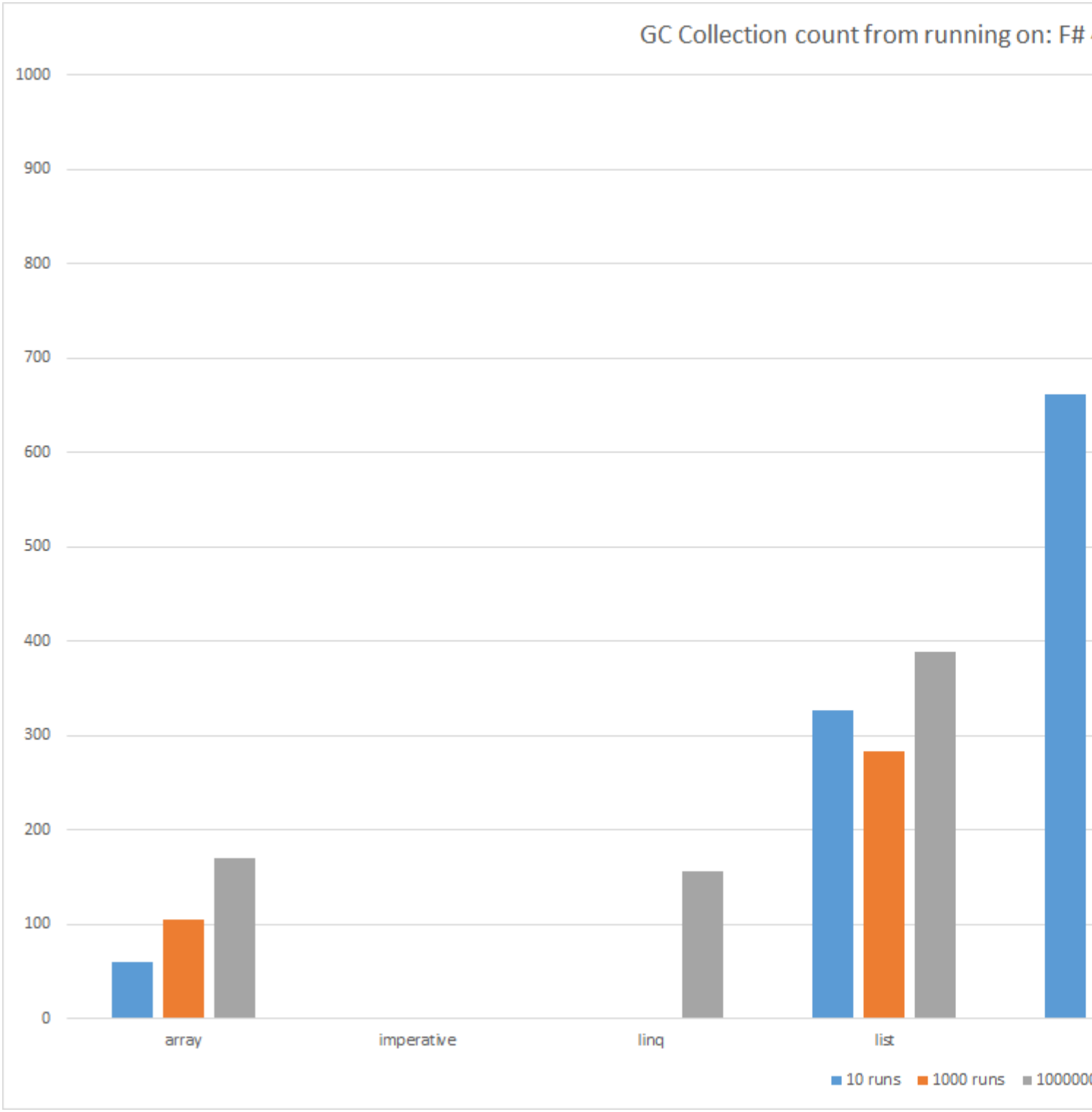
Die Balken zeigen die verstrichene Zeit an, je niedriger ist besser. Die Gesamtmenge der nützlichen Arbeit ist für alle Tests gleich, daher sind die Ergebnisse vergleichbar. Dies bedeutet

auch, dass wenige Läufe größere Datensätze implizieren.

Wie üblich beim Messen sieht man interessante Ergebnisse.

1. `List` schlechte Leistung der `List` wird mit anderen Alternativen für große Datensätze verglichen. Dies kann auf `GC` oder eine schlechte Cache-Lokalität zurückzuführen sein.
2. `Array` Leistung besser als erwartet.
3. `LINQ` besser als `Seq` . Dies ist unerwartet, da beide auf `IEnumerable<'T>` basieren. `Seq` basiert jedoch intern auf einer generischen Implementierung für alle Algorithmen, während `LINQ` spezielle Algorithmen verwendet.
4. `Push` besser als `Pull` . Dies ist zu erwarten, da die Push-Datenpipeline weniger Prüfungen aufweist
5. Die vereinfachten `Push` Datenpipelines sind mit `Nessos` vergleichbar. `Nessos` unterstützt jedoch Zug und Parallelität.
6. Bei kleinen `Nessos` verschlechtert sich die Leistung von `Nessos` möglicherweise, da der Pipeline-Setup-Overhead zunimmt.
7. Wie erwartet hat der `Imperative` Code die besten `Imperative` erzielt

Anzahl der GC-Sammlungen beim Ausführen auf: F # 4.0 - .NET 4.6.1 - x64



Die Balken zeigen die Gesamtzahl der `GC` Erfassungszählungen während des Tests an. Weniger ist besser. Dies ist ein Maß dafür, wie viele Objekte von der Datenpipeline erstellt werden.

Wie üblich beim Messen sieht man interessante Ergebnisse.

1. `List` erwartungsgemäß mehr Objekte als `Array` da eine `List` im Wesentlichen eine einzelne verknüpfte Liste von Knoten ist. Ein Array ist ein kontinuierlicher Speicherbereich.
2. Betrachten Sie die zugrunde liegenden Zahlen, und zwar sowohl `List` & `Array` 2 Generationssammlungen. Diese Art der Sammlung ist teuer.
3. `Seq` löst überraschend viele Sammlungen aus. Es ist in dieser Hinsicht überraschend noch

schlimmer als `List` .

4. `LINQ` , `Nessos` , `Push` und `Pull` löst für wenige Läufe keine Sammlungen aus. Objekte werden jedoch zugewiesen, sodass der `GC` eventuell ausgeführt werden muss.
5. Da der `Imperative` Code keine Objekte `GC` , wurden erwartungsgemäß keine `GC` Sammlungen ausgelöst.

Fazit

Alle Datenpipelines leisten in allen Testfällen die gleiche Menge an nützlicher Arbeit, aber wir sehen erhebliche Unterschiede in der Leistung und im Speicherbedarf zwischen den verschiedenen Pipelines.

Darüber hinaus stellen wir fest, dass der Overhead von Datenpipelines je nach Größe der verarbeiteten Daten unterschiedlich ist. Bei kleinen Größen `Array` zum Beispiel recht gut.

Man sollte bedenken, dass der Arbeitsaufwand in jedem Schritt der Pipeline sehr gering ist, um den Aufwand zu messen. In "echten" Situationen spielt der Aufwand für `Seq` möglicherweise keine Rolle, da die eigentliche Arbeit zeitaufwändiger ist.

Noch wichtiger sind die Unterschiede bei der Speichernutzung. `GC` ist nicht frei und es ist vorteilhaft für lange laufende Anwendungen, um den `GC` Druck zu reduzieren.

Für `F#` -Entwickler, die über Leistung und Speicherauslastung besorgt sind, wird empfohlen, Nessos Streams auszuprobieren .

Wenn Sie erstklassige Leistung benötigen, die strategisch platziert ist, ist der `Imperative` Code eine Überlegung wert.

Schließlich, wenn es um Leistung geht, machen Sie keine Annahmen. Messen und überprüfen.

Vollständiger Quellcode:

```
module PushStream =
  type Receiver<'T> = 'T -> bool
  type Stream<'T>   = Receiver<'T> -> unit

  let inline filter (f : 'T -> bool) (s : Stream<'T>) : Stream<'T> =
    fun r -> s (fun v -> if f v then r v else true)

  let inline map (m : 'T -> 'U) (s : Stream<'T>) : Stream<'U> =
    fun r -> s (fun v -> r (m v))

  let inline range b e : Stream<int> =
    fun r ->
      let rec loop i = if i <= e && r i then loop (i + 1)
      loop b

  let inline sum (s : Stream<'T>) : 'T =
    let mutable state = LanguagePrimitives.GenericZero<'T>
    s (fun v -> state <- state + v; true)
    state

module PullStream =
```

```
[<Struct>]
[<NoComparison>]
[<NoEqualityAttribute>]
type Maybe<'T>(v : 'T, hasValue : bool) =
  member    x.Value        = v
  member    x.HasValue     = hasValue
  override  x.ToString ()  =
    if hasValue then
      sprintf "Just %A" v
    else
      "Nothing"

let Nothing<'T>     = Maybe<'T> (Unchecked.defaultof<'T>, false)
let inline Just v   = Maybe<'T> (v, true)

type Iterator<'T> = unit -> Maybe<'T>
type Stream<'T>   = unit -> Iterator<'T>

let filter (f : 'T -> bool) (s : Stream<'T>) : Stream<'T> =
  fun () ->
    let i = s ()
    let rec pop () =
      let mv = i ()
      if mv.HasValue then
        let v = mv.Value
        if f v then Just v else pop ()
      else
        Nothing
    pop

let map (m : 'T -> 'U) (s : Stream<'T>) : Stream<'U> =
  fun () ->
    let i = s ()
    let pop () =
      let mv = i ()
      if mv.HasValue then
        Just (m mv.Value)
      else
        Nothing
    pop

let range b e : Stream<int> =
  fun () ->
    let mutable i = b
    fun () ->
      if i <= e then
        let p = i
        i <- i + 1
        Just p
      else
        Nothing

let inline sum (s : Stream<'T>) : 'T =
  let i = s ()
  let rec loop state =
    let mv = i ()
    if mv.HasValue then
      loop (state + mv.Value)
    else
      state
  loop LanguagePrimitives.GenericZero<'T>
```

```
module PerfTest =

  open System.Linq
#if USE_NESSOS
  open Nessos.Streams
#endif

  let now =
    let sw = System.Diagnostics.Stopwatch ()
    sw.Start ()
    fun () -> sw.ElapsedMilliseconds

  let time n a =
    let inline cc i       = System.GC.CollectionCount i

    let v                 = a ()

    System.GC.Collect (2, System.GCCollectionMode.Forced, true)

    let bcc0, bcc1, bcc2  = cc 0, cc 1, cc 2
    let b                 = now ()

    for i in 1..n do
      a () |> ignore

    let e = now ()
    let ecc0, ecc1, ecc2  = cc 0, cc 1, cc 2

    v, (e - b), ecc0 - bcc0, ecc1 - bcc1, ecc2 - bcc2

  let arrayTest n =
    Array.init (n + 1) id
    |> Array.map    int64
    |> Array.filter (fun v -> v % 2L = 0L)
    |> Array.map    ((+) 1L)
    |> Array.sum

  let imperativeTest n =
    let rec loop s i =
      if i >= 0L then
        if i % 2L = 0L then
          loop (s + i + 1L) (i - 1L)
        else
          loop s (i - 1L)
      else
        s
    loop 0L (int64 n)

  let linqTest n =
    (((Enumerable.Range(0, n + 1)).Select int64).Where(fun v -> v % 2L = 0L)).Select((+)
1L).Sum()

  let listTest n =
    List.init (n + 1) id
    |> List.map     int64
    |> List.filter  (fun v -> v % 2L = 0L)
    |> List.map     ((+) 1L)
    |> List.sum

#if USE_NESSOS
```

```
  let nessosTest n =
    Stream.initInfinite id
    |> Stream.take    (n + 1)
    |> Stream.map     int64
    |> Stream.filter  (fun v -> v % 2L = 0L)
    |> Stream.map     ((+) 1L)
    |> Stream.sum
#endif

  let pullTest n =
    PullStream.range 0 n
    |> PullStream.map     int64
    |> PullStream.filter  (fun v -> v % 2L = 0L)
    |> PullStream.map     ((+) 1L)
    |> PullStream.sum

  let pushTest n =
    PushStream.range 0 n
    |> PushStream.map     int64
    |> PushStream.filter  (fun v -> v % 2L = 0L)
    |> PushStream.map     ((+) 1L)
    |> PushStream.sum

  let seqTest n =
    Seq.init (n + 1) id
    |> Seq.map    int64
    |> Seq.filter (fun v -> v % 2L = 0L)
    |> Seq.map    ((+) 1L)
    |> Seq.sum

  let perfTest (path : string) =
    let testCases =
      [|
        "array"       , arrayTest
        "imperative"  , imperativeTest
        "linq"        , linqTest
        "list"        , listTest
        "seq"         , seqTest
#if USE_NESSOS
        "nessos"      , nessosTest
#endif
        "pull"        , pullTest
        "push"        , pushTest
      |]
    use out                   = new System.IO.StreamWriter (path)
    let write (msg : string)  = out.WriteLine msg
    let writef fmt            = FSharp.Core.Printf.kprintf write fmt

    write "Name\tTotal\tOuter\tInner\tElapsed\tCC0\tCC1\tCC2\tResult"

    let total   = 10000000
    let outers = [| 10; 1000; 1000000 |]
    for outer in outers do
      let inner = total / outer
      for name, a in testCases do
        printfn "Running %s with total=%d, outer=%d, inner=%d ..." name total outer inner
        let v, ms, cc0, cc1, cc2 = time outer (fun () -> a inner)
        printfn "  ... %d ms, cc0=%d, cc1=%d, cc2=%d, result=%A" ms cc0 cc1 cc2 v
        writef "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d" name total outer inner ms cc0 cc1 cc2 v

[<EntryPoint>]
```

```
let main argv =
  System.Environment.CurrentDirectory <- System.AppDomain.CurrentDomain.BaseDirectory
  PerfTest.perfTest "perf.tsv"
  0
```

F # Tipps und Tricks zur Leistung online lesen: https://riptutorial.com/de/fsharp/topic/3562/f-sharp-tipps-und-tricks-zur-leistung

Kapitel 11: Falten

Examples

Intro zum Falten, mit einer Handvoll Beispielen

Falten sind (Funktionen höherer Ordnung), die mit Elementsequenzen verwendet werden. Sie reduzieren `seq<'a>` zu `'b` wobei `'b` ein beliebiger Typ ist (möglicherweise noch `'a`). Dies ist ein bisschen abstrakt, also lassen Sie uns in konkrete praktische Beispiele einsteigen.

Berechnung der Summe aller Zahlen

In diesem Beispiel ist `'a` ein `int` . Wir haben eine Liste von Zahlen und wir wollen die Summe aller Zahlen berechnen. Um die Zahlen der Liste zu summieren `[1; 2; 3]` wir schreiben

```
List.fold (fun x y -> x + y) 0 [1; 2; 3] // val it : int = 6
```

Lassen Sie mich erklären, weil wir mit Listen zu tun haben , die wir verwenden `fold` in der `List` Modul, daher `List.fold` . Das erste Argument `fold` ist eine binäre Funktion, der **Ordner** . Das zweite Argument ist der **Anfangswert** . `fold` beginnt die Liste zu `fold` indem die Ordnerfunktion nacheinander auf Elemente in der Liste angewendet wird, beginnend mit dem Anfangswert und dem ersten Element. Wenn die Liste leer ist, wird der Anfangswert zurückgegeben!

Die schematische Übersicht eines Ausführungsbeispiels sieht folgendermaßen aus:

```
let add x y = x + y // this is our folder (a binary function, takes two arguments)
List.fold add 0 [1; 2; 3;]
=> List.fold add (add 0 1) [2; 3]
// the binary function is passed again as the folder in the first argument
// the initial value is updated to add 0 1 = 1 in the second argument
// the tail of the list (all elements except the first one) is passed in the third argument
// it becomes this:
List.fold add 1 [2; 3]
// Repeat untill the list is empty -> then return the "inital" value
List.fold add (add 1 2) [3]
List.fold add 3 [3] // add 1 2 = 3
List.fold add (add 3 3) []
List.fold add 6 [] // the list is empty now -> return 6
6
```

Die Funktion `List.sum` ist ungefähr `List.fold add LanguagePrimitives.GenericZero` wobei die generische Null für Ganzzahlen, Fließkommas, große Ganzzahlen usw. `List.sum` ist.

Elemente in einer Liste zählen (`count` implementieren)

Dies geschieht fast genauso wie oben, jedoch wird der tatsächliche Wert des Elements in der Liste ignoriert und stattdessen 1 hinzugefügt.

```
List.fold (fun x y -> x + 1) 0 [1; 2; 3] // val it : int = 3
```

Das kann man auch so machen:

```
[1; 2; 3]
|> List.map (fun x -> 1) // turn every elemet into 1, [1; 2; 3] becomes [1; 1; 1]
|> List.sum // sum [1; 1; 1] is 3
```

Sie können also die `count` wie folgt definieren:

```
let count xs =
    xs
    |> List.map (fun x -> 1)
    |> List.fold (+) 0 // or List.sum
```

Das Maximum der Liste finden

Diesmal verwenden wir `List.reduce` das wie `List.fold` aber ohne einen Anfangswert. In diesem Fall wissen wir nicht, was der Typ der Werte ist, die wir miteinander vergleichen:

```
let max x y = if x > y then x else y
// val max : x:'a -> y: 'a -> 'a when 'a : comparison, so only for types that we can compare
List.reduce max [1; 2; 3; 4; 5] // 5
List.reduce max ["a"; "b"; "c"] // "c", because "c" > "b" > "a"
List.reduce max [true; false] // true, because true > false
```

Das Minimum einer Liste finden

Genau wie beim Finden des Maximums unterscheidet sich der Ordner

```
let min x y = if x < y then x else y
List.reduce min [1; 2; 3; 4; 5] // 1
List.reduce min ["a"; "b"; "c"] // "a"
List.reduce min [true; false] // false
```

Listen verketten

Hier nehmen wir eine Liste der Listen auf. Die Ordnerfunktion ist der `@` -Operator

```
// [1;2] @ [3; 4] = [1; 2; 3; 4]
let merge xs ys = xs @ ys
List.fold merge [] [[1;2;3]; [4;5;6]; [7;8;9]] // [1;2;3;4;5;6;7;8;9]
```

Oder Sie können binäre Operatoren als Ordnerfunktion verwenden:

```
List.fold (@) [] [[1;2;3]; [4;5;6]; [7;8;9]] // [1;2;3;4;5;6;7;8;9]
List.fold (+) 0 [1; 2; 3] // 6
List.fold (||) false [true; false] // true, more on this below
List.fold (&&) true [true; false] // false, more on this below
// etc...
```

Berechnung der Fakultät einer Zahl

Dieselbe Idee wie beim Summieren der Zahlen, aber jetzt multiplizieren wir sie. Wenn wir die Fakultät von `n` wollen, multiplizieren wir alle Elemente in der Liste `[1 .. n]` . Code wird zu:

```
// the folder
let times x y = x * y
let factorial n = List.fold times 1 [1 .. n]
// Or maybe for big integers
let factorial n = List.fold times 1I [1I .. n]
```

Die Implementierung `forall` , `exists` und `contains`

Die Funktion `forall` prüft, ob alle Elemente einer Sequenz eine Bedingung erfüllen. `exists` überprüft , ob atleast ein Element in der Liste die Bedingung erfüllen. Zuerst müssen wir wissen, wie eine Liste von `bool` Werten `bool` . Nun, wir verwenden natürlich Kursfalten! boolesche Operatoren werden unsere Ordnerfunktionen sein.

Um zu überprüfen, ob alle Elemente in einer Liste `true` sind, `true` wir sie mit der Funktion `&&` mit dem Anfangswert `true` .

```
List.fold (&&) true [true; true; true] // true
List.fold (&&) true [] // true, empty list -> return inital value
List.fold (&&) true [false; true] // false
```

Um zu überprüfen, ob ein Element in einem Listen-Boolean `true` ist `true` reduzieren wir es mit dem `||` Operator mit `false` als Anfangswert:

```
List.fold (||) false [true; false] // true
List.fold (||) false [false; false] // false, all are false, no element is true
List.fold (||) false [] // false, empty list -> return inital value
```

Zurück zu `forall` und `exists` . Hier nehmen wir eine Liste eines beliebigen Typs, verwenden die Bedingung, um alle Elemente in boolesche Werte umzuwandeln, und reduzieren diese dann:

```
let forall condition elements =
    elements
    |> List.map condition // condition : 'a -> bool
    |> List.fold (&&) true
```

```
let exists condition elements =
    elements
    |> List.map condition
    |> List.fold (||) false
```

Um zu überprüfen, ob alle Elemente in [1; 2; 3; 4] sind kleiner als 5:

```
forall (fun n -> n < 5) [1 .. 4] // true
```

Definiere die `contains` Methode mit `exists` :

```
let contains x xs = exists (fun y -> y = x) xs
```

Oder auch

```
let contains x xs = exists ((=) x) xs
```

Prüfen Sie nun, ob die Liste [1 .. 5] den Wert 2 enthält:

```
contains 2 [1..5] // true
```

`reverse` :

```
let reverse xs = List.fold (fun acc x -> x :: acc) [] xs
reverse [1 .. 5] // [5; 4; 3; 2; 1]
```

`map` und `filter` implementieren

```
let map f = List.fold (fun acc x -> List.append acc [f x]) List.empty
// map (fun x -> x * x) [1..5] -> [1; 4; 9; 16; 25]

let filter p = Seq.fold (fun acc x -> seq { yield! acc; if p(x) then yield x }) Seq.empty
// filter (fun x -> x % 2 = 0) [1..10] -> [2; 4; 6; 8; 10]
```

Gibt es etwas, was nicht `fold` kann? Ich weiß es nicht wirklich

Berechnung der Summe aller Elemente einer Liste

Um die Summe der Terme (vom Typ float, int oder big integer) einer Zahlenliste zu berechnen, ist List.sum zu bevorzugen. In anderen Fällen ist List.fold die am besten geeignete Funktion zur Berechnung einer solchen Summe.

1. Summe der komplexen Zahlen

In diesem Beispiel deklarieren wir eine Liste komplexer Zahlen und berechnen die Summe aller Terme in der Liste.

Fügen Sie zu Beginn des Programms einen Verweis auf System.Numerics hinzu

Öffnen Sie System.Numerics

Um die Summe zu berechnen, initialisieren wir den Akkumulator mit der komplexen Zahl 0.

```
let clist = [new Complex(1.0, 52.0); new Complex(2.0, -2.0); new Complex(0.0, 1.0)]

let sum = List.fold (+) (new Complex(0.0, 0.0)) clist
```

Ergebnis:

```
(3, 51)
```

2. Summe der Nummern des Unionstyps

Angenommen, eine Liste besteht aus Nummern des Typs union (float oder int) und möchte die Summe dieser Zahlen berechnen.

Deklarieren Sie vor dem folgenden Nummerntyp:

```
type number =
| Float of float
| Int of int
```

Berechnen Sie die Summe der Typennummern einer Liste:

```
let list = [Float(1.3); Int(2); Float(10.2)]

let sum = List.fold (
                         fun acc elem ->
                                        match elem with
                                        | Float(elem) -> acc + elem
                                        | Int(elem) -> acc + float(elem)
                        ) 0.0 list
```

Ergebnis:

```
13.5
```

Der erste Parameter der Funktion, der den Akkumulator darstellt, ist vom Typ Float und der zweite Parameter, der ein Element in der Liste darstellt, hat die Typnummer. Bevor wir jedoch hinzufügen, müssen wir einen Mustervergleich verwenden und den Typ float verwenden, wenn elem vom Typ Int ist.

Falten online lesen: https://riptutorial.com/de/fsharp/topic/2250/falten

Kapitel 12: Faule Bewertung

Examples

Lazy Evaluation Einführung

Die meisten Programmiersprachen, einschließlich F #, werten die Berechnungen sofort nach einem Modell aus, das als strikte Bewertung bezeichnet wird. In Lazy Evaluation werden Berechnungen jedoch erst ausgewertet, wenn sie benötigt werden. Mit F # können wir die faule Auswertung sowohl über das `lazy` Schlüsselwort als auch über `sequences` .

```
// define a lazy computation
let comp = lazy(10 + 20)

// we need to force the result
let ans = comp.Force()
```

Wenn Sie Lazy Evaluation verwenden, werden die Ergebnisse der Berechnung zwischengespeichert. Wenn Sie also das Ergebnis nach unserer ersten Erzwingung erzwingen, wird der Ausdruck selbst nicht erneut ausgewertet

```
let rec factorial n =
  if n = 0 then
    1
  else
    (factorial (n - 1)) * n


let computation = lazy(printfn "Hello World\n"; factorial 10)

// Hello World will be printed
let ans = computation.Force()

// Hello World will not be printed here
let ansAgain = computation.Force()
```

Einführung in Lazy Evaluation in F

Wie die meisten Programmiersprachen verwendet F # standardmäßig die strikte Auswertung. Bei der strengen Auswertung werden die Berechnungen sofort ausgeführt. Im Gegensatz dazu verzögert Lazy Evaluation die Ausführung von Berechnungen, bis ihre Ergebnisse benötigt werden. Darüber hinaus werden die Ergebnisse einer Berechnung unter Lazy Evaluation zwischengespeichert, wodurch die Neubewertung eines Ausdrucks entfällt.

Wir können die Lazy-Auswertung in F # sowohl über das `lazy` Schlüsselwort als auch über `Sequences`

```
// 23 * 23 is not evaluated here
// lazy keyword creates lazy computation whose evaluation is deferred
```

```
let x = lazy(23 * 23)

// we need to force the result
let y = x.Force()

// Hello World not printed here
let z = lazy(printfn "Hello World\n"; 23424)

// Hello World printed and 23424 returned
let ans1 = z.Force()

// Hello World not printed here as z as already been evaluated, but 23424 is
// returned
let ans2 = z.Force()
```

Faule Bewertung online lesen: https://riptutorial.com/de/fsharp/topic/3682/faule-bewertung

Kapitel 13: Funktionen

Examples

Funktionen von mehr als einem Parameter

In F # haben **alle Funktionen genau einen Parameter** . Dies scheint eine merkwürdige Anweisung zu sein, da es einfach ist, mehr als einen Parameter in einer Funktionsdeklaration zu deklarieren:

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

Wenn Sie diese Funktionsdeklaration in den interaktiven F # -Interpreter eingeben, werden Sie feststellen, dass die Typensignatur wie folgt lautet:

```
val add : x:int -> y:int -> int
```

Ohne die Parameternamen lautet diese Signatur `int -> int -> int` . Der Operator `->` ist rechtsassoziativ, `int -> (int -> int)` diese Signatur entspricht `int -> (int -> int)` . Mit anderen Worten, `add` Sie ist eine Funktion , die man nimmt `int` Parameter und gibt **eine Funktion** , **die man nimmt `int` und kehrt `int`** . Versuch es:

```
let addTwo = add 2
// val addTwo : (int -> int)
let five = addTwo 3
// val five : int = 5
```

Sie können jedoch auch eine Funktion wie " `add` auf eine "konventionellere" Weise aufrufen, indem Sie direkt zwei Parameter übergeben, und es funktioniert wie erwartet:

```
let three = add 1 2
// val three : int = 3
```

Dies gilt für Funktionen mit beliebig vielen Parametern:

```
let addFiveNumbers a b c d e = a + b + c + d + e
// val addFiveNumbers : a:int -> b:int -> c:int -> d:int -> e:int -> int
let addFourNumbers = addFiveNumbers 0
// val addFourNumbers : (int -> int -> int -> int -> int)
let addThreeNumbers = addFourNumbers 0
// val addThreeNumbers : (int -> int -> int -> int)
let addTwoNumbers = addThreeNumbers 0
// val addTwoNumbers : (int -> int -> int)
let six = addThreeNumbers 1 2 3  // This will calculate 0 + 0 + 1 + 2 + 3
// val six : int = 6
```

Diese Methode, Multi-Parameter-Funktionen als Funktionen zu betrachten, die einen Parameter annehmen und neue Funktionen zurückgeben (die wiederum einen Parameter und neue

Funktionen zurückgeben können, bis Sie die letzte Funktion erreichen, die den endgültigen Parameter übernimmt und schließlich ein Ergebnis zurückgibt.) wird **Curry genannt** , zu Ehren des Mathematikers Haskell Curry, der für die Entwicklung des Konzepts berühmt ist. (Es wurde von jemand anderem erfunden, aber Curry verdient zu Recht den größten Preis dafür.)

Dieses Konzept wird in F # verwendet, und Sie sollten sich damit auskennen.

Grundlagen der Funktionen

Die meisten Funktionen in F # werden mit der `let` Syntax erstellt:

```
let timesTwo x = x * 2
```

Dies definiert eine Funktion namens `timesTwo` , die einen einzelnen Parameter `x` . Wenn Sie eine interaktive F # `fsharpi` (`fsharpi` unter OS X und Linux, `fsi.exe` unter Windows) `fsi.exe` und diese Funktion einfügen (und das `;;` hinzufügen, das `fsharpi` , den gerade eingegebenen Code auszuwerten), wird dies `fsharpi` antwortet mit:

```
val timesTwo : x:int -> int
```

Dies bedeutet, dass `timesTwo` eine Funktion ist, die einen einzelnen Parameter `x` vom Typ `int` übernimmt und ein `int` zurückgibt. Funktionssignaturen werden oft ohne die Parameternamen geschrieben. `int -> int` dieser Funktionstyp häufig als `int -> int` .

Aber warte! Woher wusste F #, dass `x` ein Integer-Parameter war, da Sie den Typ nie angegeben haben? Das liegt am **Typ Inferenz** . Da Sie im Funktionshauptteil `x` mit `2` multipliziert haben, müssen die Typen von `x` und `2` gleich sein. (Generell gilt, dass F # Werte nicht implizit in andere Typen umwandelt; Sie müssen explizit alle gewünschten Typumwandlungen angeben.)

Wenn Sie eine Funktion erstellen möchten, für die keine Parameter erforderlich sind, ist dies der **falsche** Weg:

```
let hello =  // This is a value, not a function
    printfn "Hello world"
```

Der **richtige** Weg, es zu tun, ist:

```
let hello () =
    printfn "Hello world"
```

Diese `hello` Funktion hat den Typ `unit -> unit` , was im Typ "Einheit" erläutert wird.

Curried vs getippte Funktionen

Es gibt zwei Möglichkeiten, Funktionen mit mehreren Parametern in F #, Curried-Funktionen und getopften Funktionen zu definieren.

```
let curriedAdd x y = x + y // Signature: x:int -> y:int -> int
```

```
let tupledAdd (x, y) = x + y // Signature:  x:int * y:int -> int
```

Alle Funktionen, die von außerhalb von F # definiert wurden (z. B. das .NET-Framework), werden in F # mit dem getippten Formular verwendet. Die meisten Funktionen in F # -Kernmodulen werden in Curried-Form verwendet.

Die Curry-Form wird als idiomatische F # betrachtet, da sie eine teilweise Anwendung erlaubt. Keines der folgenden zwei Beispiele ist mit der getupften Form möglich.

```
let addTen = curriedAdd 10 // Signature: int -> int

// Or with the Piping operator
3 |> curriedAdd 7 // Evaluates to 10
```

Der Grund dafür ist, dass die Funktion Curried eine Funktion zurückgibt, wenn sie mit einem Parameter aufgerufen wird. Willkommen bei der funktionalen Programmierung!

```
let explicitCurriedAdd x = (fun y -> x + y) // Signature: x:int -> y:int -> int
let veryExplicitCurriedAdd = (fun x -> (fun y -> x + y)) // Same signature
```

Sie können sehen, dass es genau dieselbe Signatur ist.

Wenn Sie jedoch mit anderem .NET-Code kommunizieren, wie beim Schreiben von Bibliotheken, ist es wichtig, Funktionen mithilfe des getupten Formulars zu definieren.

Inlining

Durch Inlining können Sie einen Aufruf einer Funktion durch den Hauptteil der Funktion ersetzen.

Dies ist manchmal aus Leistungsgründen im kritischen Teil des Codes hilfreich. Das Gegenstück ist jedoch, dass der Aufbau Ihrer Assembly viel Platz beansprucht, da der Rumpf der Funktion überall dort dupliziert wird, wo ein Aufruf stattgefunden hat. Sie müssen vorsichtig sein, wenn Sie entscheiden, ob Sie eine Funktion integrieren möchten oder nicht.

Eine Funktion kann mit dem `inline` Schlüsselwort eingebettet werden:

```
// inline indicate that we want to replace each call to sayHello with the body
// of the function
let inline sayHello s1 =
    sprintf "Hello %s" s1

// Call to sayHello will be replaced with the body of the function
let s = sayHello "Foo"
// After inlining ->
// let s = sprintf "Hello %s" "Foo"

printfn "%s" s
// Output
// "Hello Foo"
```

Ein weiteres Beispiel mit lokalem Wert:

```
let inline addAndMulti num1 num2 =
    let add = num1 + num2
    add * 2

let i = addAndMulti 2 2
// After inlining ->
// let add = 2 + 2
// let i = add * 2

printfn "%i" i
// Output
// 8
```

Pipe vorwärts und rückwärts

Über Pipe-Operatoren werden Parameter auf einfache und elegante Weise an eine Funktion übergeben. Dadurch können Zwischenwerte eliminiert und Funktionsaufrufe leichter lesbar gemacht werden.

In F # gibt es zwei Pipe-Operatoren:

- **Forward** (`|>`): Parameter werden von links nach rechts übergeben

```
let print message =
    printf "%s" message

// "Hello World" will be passed as a parameter to the print function
"Hello World" |> print
```

- **Rückwärts** (`<|`): Parameter von rechts nach links übergeben

```
let print message =
    printf "%s" message

// "Hello World" will be passed as a parameter to the print function
print <| "Hello World"
```

Hier ist ein Beispiel ohne Pipe-Operator:

```
// We want to create a sequence from 0 to 10 then:
// 1 Keep only even numbers
// 2 Multiply them by 2
// 3 Print the number

let mySeq = seq { 0..10 }
let filteredSeq = Seq.filter (fun c -> (c % 2) = 0) mySeq
let mappedSeq = Seq.map ((*) 2) filteredSeq
let finalSeq = Seq.map (sprintf "The value is %i.") mappedSeq

printfn "%A" finalSeq
```

Wir können das vorherige Beispiel verkürzen und mit dem Forward-Pipe-Operator sauberer gestalten:

```
// Using forward pipe, we can eliminate intermediate let binding
let finalSeq =
    seq { 0..10 }
    |> Seq.filter (fun c -> (c % 2) = 0)
    |> Seq.map ((*) 2)
    |> Seq.map (sprintf "The value is %i.")

printfn "%A" finalSeq
```

Jedes Funktionsergebnis wird als Parameter an die nächste Funktion übergeben.

Wenn Sie mehrere Parameter an den Pipe-Operator übergeben möchten, müssen Sie ein | hinzufügen für jeden zusätzlichen Parameter und erstellen Sie ein Tuple mit den Parametern. Der native F # -Pipe-Operator unterstützt bis zu drei Parameter (|||> oder <|||).

```
let printPerson name age =
    printf "My name is %s, I'm %i years old" name age

("Foo", 20) ||> printPerson
```

Funktionen online lesen: https://riptutorial.com/de/fsharp/topic/2525/funktionen

Kapitel 14: Geben Sie Anbieter ein

Examples

Verwendung des CSV-Typanbieters

Angesichts der folgenden CSV-Datei:

```
Id,Name
1,"Joel"
2,"Adam"
3,"Ryan"
4,"Matt"
```

Sie können die Daten mit folgendem Skript lesen:

```
#r "FSharp.Data.dll"
open FSharp.Data

type PeopleDB = CsvProvider<"people.csv">

let people = PeopleDB.Load("people.csv") // this can be a URL

let joel = people.Rows |> Seq.head

printfn "Name: %s, Id: %i" joel.Name joel.Id
```

Verwenden des WMI-Typanbieters

Mit dem WMI-Typanbieter können Sie WMI-Services mit starker Typisierung abfragen.

Um die Ergebnisse einer WMI-Abfrage als JSON auszugeben,

```
open FSharp.Management
open Newtonsoft.Json

// `Local` is based off of the WMI available at localhost.
type Local = WmiProvider<"localhost">

let data =
    [for d in Local.GetDataContext().Win32_DiskDrive -> d.Name, d.Size]

printfn "%A" (JsonConvert.SerializeObject data)
```

Geben Sie Anbieter ein online lesen: https://riptutorial.com/de/fsharp/topic/1631/geben-sie-anbieter-ein

Kapitel 15: Generics

Examples

Umkehrung einer Liste eines beliebigen Typs

Um eine Liste umzukehren, ist es nicht wichtig, um welchen Typ es sich bei den Listenelementen handelt, sondern nur um die Reihenfolge, in der sie sich befinden. Dies ist der perfekte Kandidat für eine generische Funktion, sodass dieselbe Reverseal-Funktion unabhängig von der übergebenen Liste verwendet werden kann.

```
let rev list =
    let rec loop acc = function
        | []           -> acc
        | head :: tail -> loop (head :: acc) tail
    loop [] list
```

Der Code macht keine Annahmen über die Arten der Elemente. Der Compiler (oder F # interactive) teilt Ihnen mit, dass die Typunterschrift dieser Funktion `'T list -> 'T list` . Das `'T` sagt Ihnen, dass es sich um einen generischen Typ ohne Einschränkungen handelt. Möglicherweise wird auch `'a` anstelle von `'T` angezeigt. Der Buchstabe ist unwichtig, da er nur ein *allgemeiner* Platzhalter ist. Wir können eine `int list` oder eine `string list` , und beide funktionieren erfolgreich und geben eine `int list` bzw. eine `string list` zurück.

Zum Beispiel in F # interactive:

```
> let rev list = ...
val it : 'T list -> 'T list
> rev [1; 2; 3; 4];;
val it : int list = [4; 3; 2; 1]
> rev ["one", "two", "three"];;
val it : string list = ["three", "two", "one"]
```

Eine Liste einem anderen Typ zuordnen

```
let map f list =
    let rec loop acc = function
        | []           -> List.rev acc
        | head :: tail -> loop (f head :: acc) tail
    loop [] list
```

Die Signatur dieser Funktion ist `('a -> 'b) -> 'a list -> 'b list` Dies ist die generischste. Dies verhindert nicht, dass `'a` derselbe Typ ist wie `'b` , aber es lässt auch zu, dass sie verschieden sind. Hier können Sie sehen , dass das `'a` Typ, der die Parameter der Funktion ist `f` müssen die Art des Spiels `list` Parameter. Diese Funktion ist immer noch generisch, es gibt jedoch einige Einschränkungen für die Eingaben. Wenn die Typen nicht übereinstimmen, tritt ein Kompilierungsfehler auf.

Beispiele:

```
> let map f list = ...
val it : ('a -> 'b) -> 'a list -> 'b list
> map (fun x -> float x * 1.5) [1; 2; 3; 4];;
val it : float list = [1.5; 3.0; 4.5; 6.0]
> map (sprintf "abc%.1f") [1.5; 3.0; 4.5; 6.0];;
val it : string list = ["abc1.5"; "abc3.0"; "abc4.5"; "abc6.0"]
> map (fun x -> x + 1) [1.0; 2.0; 3.0];;
error FS0001: The type 'float' does not match the type 'int'
```

Generics online lesen: https://riptutorial.com/de/fsharp/topic/7731/generics

Kapitel 16: Klassen

Examples

Klasse deklarieren

```
// class declaration with a list of parameters for a primary constructor
type Car (model: string, plates: string, miles: int) =

    // secondary constructor (it must call primary constructor)
    new (model, plates) =
        let miles = 0
        new Car(model, plates, miles)

    // fields
    member this.model = model
    member this.plates = plates
    member this.miles = miles
```

Instanz erstellen

```
let myCar = Car ("Honda Civic", "blue", 23)
// or more explicitly
let (myCar : Car) = new Car ("Honda Civic", "blue", 23)
```

Klassen online lesen: https://riptutorial.com/de/fsharp/topic/3003/klassen

Kapitel 17: Listen

Syntax

- [] // eine leere Liste.

  head :: tail // eine Konstruktionszelle, die ein Element, head und eine list, tail enthält. :: heißt Cons-Operator.

  lassen Sie list1 = [1; 2; 3] // Beachten Sie die Verwendung eines Semikolons.

  let list2 = 0 :: list1 // Das Ergebnis ist [0; 1; 2; 3]

  let list3 = list1 @ list2 // Das Ergebnis ist [1; 2; 3; 0; 1; 2; 3]. @ ist der Append-Operator.

  let list4 = [1..3] // Ergebnis ist [1; 2; 3]

  let list5 = [1..2..10] // Ergebnis ist [1; 3; 5; 7; 9]

  Lassen Sie list6 = [für i in 1..10 tun, wenn i% 2 = 1, dann ergibt i] // Ergebnis ist [1; 3; 5; 7; 9]

Examples

Grundlegende Listennutzung

```
let list1 = [ 1; 2 ]
let list2 = [ 1 .. 100 ]

// Accessing an element
printfn "%A" list1.[0]

// Pattern matching
let rec patternMatch aList =
    match aList with
    | [] -> printfn "This is an empty list"
    | head::tail -> printfn "This list consists of a head element %A and a tail list %A" head
tail
                    patternMatch tail

patternMatch list1

// Mapping elements
let square x = x*x
let list2squared = list2
                   |> List.map square
printfn "%A" list2squared
```

Berechnung der Gesamtsumme der Zahlen in einer Liste

Durch Rekursion

```
let rec sumTotal list =
    match list with
    | [] -> 0 // empty list -> return 0
    | head :: tail -> head + sumTotal tail
```

Das obige Beispiel sagt: "Schauen Sie sich die `list` , ist sie leer? Return 0. Ansonsten handelt es sich um eine nicht leere Liste. Es könnte sich also um `[1]` , `[1; 2]` , [1; 2; 3] usw. Wenn `list` [1] ist, binden Sie die Variable `head` an `1` und `tail` an `[]` und führen Sie `head + sumTotal tail` .

Beispielausführung:

```
sumTotal [1; 2; 3]
// head -> 1, tail -> [2; 3]
1 + sumTotal [2; 3]
1 + (2 + sumTotal [3])
1 + (2 + (3 + sumTotal [])) // sumTotal [] is defined to be 0, recursion stops here
1 + (2 + (3 + 0)))
1 + (2 + 3)
1 + 5
6
```

Ein allgemeinerer Weg, das obige Muster einzukapseln, ist die Verwendung von Funktionsfalten! `sumTotal` wird so:

```
let sumTotal list = List.fold (+) 0 list
```

Listen erstellen

Eine Liste zum Erstellen einer Liste besteht darin, Elemente in zwei eckige Klammern zu setzen, die durch Semikola getrennt sind. Die Elemente müssen den gleichen Typ haben.

Beispiel:

```
> let integers = [1; 2; 45; -1];;
val integers : int list = [1; 2; 45; -1]


> let floats = [10.7; 2.0; 45.3; -1.05];;
val floats : float list = [10.7; 2.0; 45.3; -1.05]
```

Wenn eine Liste kein Element enthält, ist sie leer. Eine leere Liste kann wie folgt deklariert werden:

```
> let emptyList = [];;
val emptyList : 'a list
```

Anderes Beispiel

Um eine Liste von Bytes zu erstellen, einfach die Ganzzahlen umwandeln:

```
> let bytes = [byte(55); byte(10); byte(100)];;
```

```
val bytes : byte list = [55uy; 10uy; 100uy]
```

Es ist auch möglich, Funktionslisten, Elemente eines zuvor definierten Typs, Objekte einer Klasse usw. zu definieren.

Beispiel

```
> type number = | Real of float | Integer of int;;

type number =
  | Real of float
  | Integer of int

> let numbers = [Integer(45); Real(0.0); Integer(127)];;
val numbers : number list = [Integer 45; Real 0.0; Integer 127]
```

Bereiche

Für bestimmte Elementtypen (int, float, char, ...) ist es möglich, eine Liste nach dem Startelement und dem Endelement zu definieren, wobei die folgende Vorlage verwendet wird:

```
[start..end]
```

Beispiele:

```
> let c=['a' .. 'f'];;
val c : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']

let f=[45 .. 60];;
val f : int list =
  [45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60]
```

Sie können auch einen Schritt für bestimmte Typen mit dem folgenden Modell angeben:

```
[start..step..end]
```

Beispiele:

```
> let i=[4 .. 2 .. 11];;
val i : int list = [4; 6; 8; 10]

> let r=[0.2 .. 0.05 .. 0.28];;
val r : float list = [0.2; 0.25]
```

Generator

Eine andere Möglichkeit zum Erstellen einer Liste besteht darin, sie automatisch mithilfe des Generators zu generieren.

Wir können eines der folgenden Modelle verwenden:

```
[for <identifier> in range -> expr]
```

oder

```
[for <identifier> in range do ... yield expr]
```

Beispiele

```
> let oddNumbers = [for i in 0..10 -> 2 * i + 1];; // odd numbers from 1 to 21
val oddNumbers : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19; 21]

> let multiples3Sqrt = [for i in 1..27 do if i % 3 = 0 then yield sqrt(float(i))];; //sqrt of
multiples of 3 from 3 to 27
val multiples3Sqrt : float list =
  [1.732050808; 2.449489743; 3.0; 3.464101615; 3.872983346; 4.242640687;    4.582575695;
4.898979486; 5.196152423]
```

Operatoren

Einige Operatoren können zum Erstellen von Listen verwendet werden:

Nachteile operator:

Dieser Operator :: wird verwendet, um einer Liste ein Kopfelement hinzuzufügen:

```
> let l=12::[] ;;
val l : int list = [12]

> let l1=7::[14; 78; 0] ;;
val l1 : int list = [7; 14; 78; 0]

> let l2 = 2::3::5::7::11::[13;17] ;;
val l2 : int list = [2; 3; 5; 7; 11; 13; 17]
```

Verkettung

Die Verkettung von Listen erfolgt mit dem Operator @.

```
> let l1 = [12.5;89.2];;
val l1 : float list = [12.5; 89.2]

> let l2 = [1.8;7.2] @ l1;;
val l2 : float list = [1.8; 7.2; 12.5; 89.2]
```

Listen online lesen: https://riptutorial.com/de/fsharp/topic/1268/listen

Kapitel 18: Maßeinheiten

Bemerkungen

Einheiten zur Laufzeit

Maßeinheiten werden nur für die statische Überprüfung durch den Compiler verwendet und sind zur Laufzeit nicht verfügbar. Sie können nicht in Reflektionen oder in Methoden wie `ToString` .

Beispielsweise gibt C # ein `double` ohne Einheiten für ein Feld des Typs `float<m>` das durch eine F # -Bibliothek definiert und aus dieser freigegeben wird.

Examples

Sicherstellung konsistenter Einheiten in Berechnungen

Maßeinheiten sind zusätzliche Typanmerkungen, die zu Gleitkommazahlen oder Ganzzahlen hinzugefügt werden können. Sie können verwendet werden, um während der Kompilierzeit zu überprüfen, ob die Berechnungen Einheiten einheitlich verwenden.

Anmerkungen definieren:

```
[<Measure>] type m // meters
[<Measure>] type s // seconds
[<Measure>] type accel = m/s^2 // acceleration defined as meters per second squared
```

Einmal definiert, können Anmerkungen verwendet werden, um zu überprüfen, ob ein Ausdruck den erwarteten Typ ergibt.

```
// Compile-time checking that this function will return meters, since (m/s^2) * (s^2) -> m
// Therefore we know units were used properly in the calculation.
let freeFallDistance (time:float<s>) : float<m> =
    0.5 * 9.8<accel> * (time*time)

// It is also made explicit at the call site, so we know that the parameter passed should be
in seconds
let dist:float<m> = freeFallDistance 3.0<s>
printfn "%f" dist
```

Umrechnungen zwischen Einheiten

```
[<Measure>] type m // meters
[<Measure>] type cm // centimeters

// Conversion factor
let cmInM = 100<cm/m>
```

```
let distanceInM = 1<m>
let distanceInCM = distanceInM * cmInM // 100<cm>

// Conversion function
let cmToM (x : int<cm>) = x / 100<cm/m>
let mToCm (x : int<m>) = x * 100<cm/m>

cmToM 100<cm> // 1<m>
mToCm 1<m> // 100<cm>
```

Beachten Sie, dass der F # -Compiler nicht weiß, dass `1<m> 100<cm>` . Die Einheiten sind, soweit es wichtig ist, separate Typen. Sie können ähnliche Funktionen schreiben, um von Metern in Kilogramm zu konvertieren, und der Compiler würde das nicht interessieren.

```
[<Measure>] type kg

// Valid code, invalid physics
let kgToM x = x / 100<kg/m>
```

Es ist nicht möglich, Maßeinheiten als Vielfache anderer Einheiten wie z. B. zu definieren

```
// Invalid code
[<Measure>] type m = 100<cm>
```

Einheiten "pro etwas" zu definieren, zum Beispiel Hertz, das Messen der Frequenz ist einfach "pro Sekunde", ist ziemlich einfach.

```
// Valid code
[<Measure>] type s
[<Measure>] type Hz = /s

1 / 1<s> = 1 <Hz> // Evaluates to true

[<Measure>] type N = kg m/s // Newtons, measuring force. Note lack of multiplication sign.

// Usage
let mass = 1<kg>
let distance = 1<m>
let time = 1<s>

let force = mass * distance / time // Evaluates to 1<kg m/s>
force = 1<N> // Evaluates to true
```

Verwendung von LanguagePrimitives zum Erhalten oder Einstellen von Einheiten

Wenn eine Funktion Einheiten aufgrund untergeordneter Operationen nicht automatisch speichert, können Sie das Modul `LanguagePrimitives` verwenden, um Einheiten für die Grundelemente festzulegen, die sie unterstützen:

```
/// This cast preserves units, while changing the underlying type
let inline castDoubleToSingle (x : float<'u>) : float32<'u> =
    LanguagePrimitives.Float32WithMeasure (float32 x)
```

Um Maßeinheiten einem Gleitkommawert mit doppelter Genauigkeit zuzuordnen, multiplizieren Sie einfach mit den korrekten Einheiten mit eins:

```
[<Measure>]
type USD

let toMoneyImprecise (amount : float) =
   amount * 1.<USD>
```

Um Maßeinheiten einem Wert ohne Einheiten zuzuweisen, der nicht System.Double ist, z. B. aus einer Bibliothek, die in einer anderen Sprache geschrieben wurde, verwenden Sie eine Konvertierung:

```
open LanguagePrimitives

let toMoney amount =
   amount |> DecimalWithMeasure<'u>
```

Hier sind die von F # interactive gemeldeten Funktionstypen:

```
val toMoney : amount:decimal -> decimal<'u>
val toMoneyImprecise : amount:float -> float<USD>
```

Parameter der Maßeinheitenart

Das Attribut `[<Measure>]` kann für Typparameter verwendet werden, um generische Typen in Bezug auf Maßeinheiten zu deklarieren:

```
type CylinderSize<[<Measure>] 'u> =
    { Radius : float<'u>
      Height : float<'u> }
```

Testnutzung:

```
open Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols

/// This has type CylinderSize<m>.
let testCylinder =
    { Radius = 14.<m>
      Height =  1.<m> }
```

Verwenden Sie standardisierte Gerätetypen, um die Kompatibilität zu gewährleisten

In der F # -Kernbibliothek in `Microsoft.FSharp.Data.UnitSystems.SI` beispielsweise Typen für SI-Einheiten standardisiert. Öffnen Sie den entsprechenden Sub-Namespace, `UnitNames` oder `UnitSymbols` , um diese zu verwenden. Wenn nur wenige SI-Einheiten erforderlich sind, können sie mit Typ-Aliasnamen importiert werden:

```
/// Seconds, the SI unit of time. Type abbreviation for the Microsoft standardized type.
type [<Measure>] s = Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols.s
```

Einige Benutzer neigen dazu, Folgendes zu tun, was **nicht gemacht werden sollte,** wenn bereits eine Definition verfügbar ist:

```
/// Seconds, the SI unit of time
type [<Measure>] s // DO NOT DO THIS! THIS IS AN EXAMPLE TO EXPLAIN A PROBLEM.
```

Der Unterschied wird beim Anschluss an anderen Code deutlich, der sich auf die Standard-SI-Typen bezieht. Code, der sich auf die Standardeinheiten bezieht, ist kompatibel, während Code, der seinen eigenen Typ definiert, nicht mit Code kompatibel ist, der seine spezifische Definition nicht verwendet.

Verwenden Sie daher immer die Standardtypen für SI-Einheiten. Es spielt keine Rolle, ob Sie sich auf `UnitNames` oder `UnitSymbols` beziehen, da sich gleichwertige Namen in diesen beiden auf denselben Typ beziehen:

```
open Microsoft.FSharp.Data.UnitSystems.SI

/// This is valid, since both versions refer to the same authoritative type.
let validSubtraction = 1.<UnitSymbols.s> - 0.5<UnitNames.second>
```

Maßeinheiten online lesen: https://riptutorial.com/de/fsharp/topic/1055/ma-einheiten

Kapitel 19: Memoization

Examples

Einfaches Memo

Memoization besteht aus Caching-Funktionsergebnissen, um zu vermeiden, dass dasselbe Ergebnis mehrmals berechnet wird. Dies ist nützlich, wenn Sie mit Funktionen arbeiten, die kostspielige Berechnungen durchführen.

Wir können eine einfache Faktorfunktion als Beispiel verwenden:

```
let factorial index =
    let rec innerLoop i acc =
        match i with
        | 0 | 1 -> acc
        | _ -> innerLoop (i - 1) (acc * i)

    innerLoop index 1
```

Ein mehrfacher Aufruf dieser Funktion mit demselben Parameter führt immer wieder zu derselben Berechnung.

Memoization hilft uns, das faktorielle Ergebnis zwischenzuspeichern und zurückzugeben, wenn der gleiche Parameter erneut übergeben wird.

Hier ist eine einfache Memo-Implementierung:

```
// memoization takes a function as a parameter
// This function will be called every time a value is not in the cache
let memoization f =
    // The dictionary is used to store values for every parameter that has been seen
    let cache = Dictionary<_,_>()
    fun c ->
        let exist, value = cache.TryGetValue (c)
        match exist with
        | true ->
            // Return the cached result directly, no method call
            printfn "%O -> In cache" c
            value
        | _ ->
            // Function call is required first followed by caching the result for next call
with the same parameters
            printfn "%O -> Not in cache, calling function..." c
            let value = f c
            cache.Add (c, value)
            value
```

Die `memoization` Funktion übernimmt einfach eine Funktion als Parameter und gibt eine Funktion mit derselben Signatur zurück. Sie könnten dies in der Methodensignatur `f:('a -> 'b) -> ('a -> 'b)` . Auf diese Weise können Sie die Memoisierung auf dieselbe Weise verwenden, als würden

Sie die factorial-Methode aufrufen.

Die `printfn` Aufrufe sollen zeigen, was passiert, wenn wir die Funktion mehrmals aufrufen. Sie können sicher entfernt werden.

Die Verwendung von Memoization ist einfach:

```
// Pass the function we want to cache as a parameter via partial application
let factorialMem = memoization factorial

// Then call the memoized function
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 4)
printfn "%i" (factorialMem 4)

// Prints
// 10 -> Not in cache, calling function...
// 3628800
// 10 -> In cache
// 3628800
// 10 -> In cache
// 3628800
// 4 -> Not in cache, calling function...
// 24
// 4 -> In cache
// 24
```

Memoisierung in einer rekursiven Funktion

In dem vorherigen Beispiel der Berechnung der Fakultät einer Ganzzahl geben Sie alle in der Rekursion berechneten Werte der Fakultät in die Hashtabelle ein, die nicht in der Tabelle erscheinen.

Wie im Artikel zum Memoisieren deklarieren wir eine Funktion `f` , die einen Funktionsparameter- `fact` und einen Ganzzahl-Parameter akzeptiert. Diese Funktion `f` enthält Anweisungen zur Berechnung der Fakultät von `n` aus `fact(n-1)` .

Dies ermöglicht es, Rekursion durch die von `memorec` zurückgegebene `memorec` und nicht `fact` selbst `memorec` und möglicherweise die Berechnung zu stoppen, wenn der `memorec fact(n-1)` bereits berechnet wurde und sich in der Hash-Tabelle befindet.

```
let memorec f =
   let cache = Dictionary<_,_>()
   let rec frec n =
       let value = ref 0
       let exist = cache.TryGetValue(n, value)
       match exist with
       | true ->
           printfn "%O -> In cache" n
       | false ->
           printfn "%O -> Not in cache, calling function..." n
           value := f frec n
           cache.Add(n, !value)
```

```
        !value
    in frec

let f = fun fact n -> if n<2 then 1 else n*fact(n-1)

[<EntryPoint>]
let main argv =
    let g = memorec(f)
    printfn "%A" (g 10)
    printfn "%A" "----------------------"
    printfn "%A" (g 5)
    Console.ReadLine()
    0
```

Ergebnis:

```
10 -> Not in cache, calling function...
9 -> Not in cache, calling function...
8 -> Not in cache, calling function...
7 -> Not in cache, calling function...
6 -> Not in cache, calling function...
5 -> Not in cache, calling function...
4 -> Not in cache, calling function...
3 -> Not in cache, calling function...
2 -> Not in cache, calling function...
1 -> Not in cache, calling function...
3628800
"----------------------"
5 -> In cache
120
```

Memoization online lesen: https://riptutorial.com/de/fsharp/topic/2698/memoization

Kapitel 20: Monaden

Examples

Monaden verstehen kommt aus der Praxis

Dieses Thema richtet sich an fortgeschrittene F # -Entwickler

"Was sind Monaden?" ist eine häufig gestellte Frage. Dies ist leicht zu beantworten, aber wie in Hitchhikers 'Leitfaden zur Galaxie ist uns klar, dass wir die Antwort nicht verstehen, weil wir nicht wussten, wonach wir gefragt haben.

Viele glauben, dass der Weg zum Verständnis der Monaden darin besteht, sie zu praktizieren. Als Programmierer interessieren wir uns normalerweise nicht für die mathematische Grundlage für das Substitutionsprinzip, die Untertypen oder Unterklassen von Liskov. Durch die Verwendung dieser Ideen haben wir eine Intuition für das, was sie repräsentieren, erworben. Gleiches gilt für Monaden.

Um Ihnen den Einstieg in Monaden zu erleichtern, zeigt dieses Beispiel, wie Sie eine Monadic Parser Combinator- Bibliothek erstellen . Dies kann Ihnen zwar beim Einstieg helfen, das Verständnis wird jedoch dadurch entstehen, dass Sie Ihre eigene Monadic-Bibliothek schreiben.

Genug Prosa, Zeit für Code

Der Parser-Typ:

```
// A Parser<'T> is a function that takes a string and position
//  and returns an optionally parsed value and a position
//  A parsed value means the position points to the character following the parsed value
//  No parsed value indicates a parse failure at the position
type Parser<'T> = Parser of (string*int -> 'T option*int)
```

Mit dieser Definition eines Parsers definieren wir einige grundlegende Parserfunktionen

```
// Runs a parser 't' on the input string 's'
let run t s =
  let (Parser tps) = t
  tps (s, 0)

// Different ways to create parser result
let succeedWith v p = Some v, p
let failAt        p = None  , p

// The 'satisfy' parser succeeds if the character at the current position
//  passes the 'sat' function
let satisfy sat : Parser<char> = Parser <| fun (s, p) ->
  if p < s.Length && sat s.[p] then succeedWith s.[p] (p + 1)
  else failAt p

// 'eos' succeeds if the position is beyond the last character.
//  Useful when testing if the parser have consumed the full string
```

```
let eos : Parser<unit> = Parser <| fun (s, p) ->
  if p < s.Length then failAt p
  else succeedWith () p

let anyChar       = satisfy (fun _ -> true)
let char ch       = satisfy ((=) ch)
let digit         = satisfy System.Char.IsDigit
let letter        = satisfy System.Char.IsLetter
```

`satisfy` , ist eine Funktion , die eine gegebene `sat` Funktion erzeugt einen Parser, wenn wir nicht bestanden erfolgreich `EOS` und das Zeichen an der gegenwärtigen Position die Pässe `sat` Funktion. Mit `satisfy` wir eine Reihe von nützlichen Charakter Parser erstellen.

Ausführen in FSI:

```
> run digit "";;
val it : char option * int = (null, 0)
> run digit "123";;
val it : char option * int = (Some '1', 1)
> run digit "hello";;
val it : char option * int = (null, 0)
```

Wir haben einige grundlegende Parser installiert. Wir werden sie mit Parser-Kombinatorfunktionen zu leistungsfähigeren Parsern kombinieren

```
// 'fail' is a parser that always fails
let fail<'T>      = Parser <| fun (s, p) -> failAt p
// 'return_' is a parser that always succeed with value 'v'
let return_ v     = Parser <| fun (s, p) -> succeedWith v p

// 'bind' let us combine two parser into a more complex parser
let bind t uf     = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    -> None, tp
  | Some tv ->
    let u = uf tv
    let (Parser ups) = u
    ups (s, tp)
```

Die Namen und Signaturen werden nicht willkürlich ausgewählt, aber wir werden darauf nicht näher eingehen. Stattdessen wollen wir sehen, wie wir mit `bind` Parser zu komplexeren kombinieren:

```
> run (bind digit (fun v -> digit)) "123";;
val it : char option * int = (Some '2', 2)
> run (bind digit (fun v -> bind digit (fun u -> return_ (v,u)))) "123";;
val it : (char * char) option * int = (Some ('1', '2'), 2)
> run (bind digit (fun v -> bind digit (fun u -> return_ (v,u)))) "1";;
val it : (char * char) option * int = (null, 1)
```

Das zeigt uns, dass `bind` es uns erlaubt, zwei Parser zu einem komplexeren Parser zu kombinieren. Als Ergebnis von `bind` entsteht ein Parser, der wiederum wieder kombiniert werden

kann.

```
> run (bind digit (fun v -> bind digit (fun w -> bind digit (fun u -> return_ (v,w,u)))))
"123";;
val it : (char * char * char) option * int = (Some ('1', '2', '3'), 3)
```

`bind` wird die grundlegende Möglichkeit sein, Parser zu kombinieren, obwohl wir Hilfsfunktionen definieren werden, um die Syntax zu vereinfachen.

Eine der Dinge, die die Syntax vereinfachen können, sind Berechnungsausdrücke . Sie sind leicht zu definieren:

```
type ParserBuilder() =
  member x.Bind       (t, uf) = bind      t   uf
  member x.Return     v       = return_   v
  member x.ReturnFrom t       = t

// 'parser' enables us to combine parsers using 'parser { ... }' syntax
let parser = ParserBuilder()
```

FSI

```
let p = parser {
          let! v = digit
          let! u = digit
          return v,u
        }
run p "123"
val p : Parser<char * char> = Parser <fun:bind@49-1>
val it : (char * char) option * int = (Some ('1', '2'), 2)
```

Das ist äquivalent zu:

```
> let p = bind digit (fun v -> bind digit (fun u -> return_ (v,u)))
run p "123";;
val p : Parser<char * char> = Parser <fun:bind@49-1>
val it : (char * char) option * int = (Some ('1', '2'), 2)
```

Ein weiterer grundlegender Parser-Kombinator, den wir alot verwenden werden, ist `orElse` :

```
// 'orElse' creates a parser that runs parser 't' first, if that is successful
//  the result is returned otherwise the result of parser 'u' is returned
let orElse t u    = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    ->
    let (Parser ups) = u
    ups (s, p)
  | Some tv -> succeedWith tv tp
```

Dadurch können wir `letterOrDigit` definieren:

```
> let letterOrDigit = orElse letter digit;;
val letterOrDigit : Parser<char> = Parser <fun:orElse@70-1>
> run letterOrDigit "123";;
val it : char option * int = (Some '1', 1)
> run letterOrDigit "hello";;
val it : char option * int = (Some 'h', 1)
> run letterOrDigit "!!!";;
val it : char option * int = (null, 0)
```

Ein Hinweis zu Infix-Operatoren

Ein häufiges Problem bei FP ist die Verwendung ungewöhnlicher Infix-Operatoren wie `>>=` , `>=>` , `<-` und so weiter. Die meisten machen sich jedoch keine Gedanken über die Verwendung von `+` , `-` , `*` , `/` und `%` . Dies sind bekannte Operatoren, die zum Zusammenstellen von Werten verwendet werden. Ein großer Teil von FP besteht jedoch darin, nicht nur Werte zu erstellen, sondern auch zu funktionieren. Für einen fortgeschrittenen FP-Entwickler sind die Infix-Operatoren `>>=` , `>=>` , `<-` bekannt und sollten über spezifische Signaturen sowie Semantik verfügen.

Für die bisher definierten Funktionen definieren wir die folgenden Infix-Operatoren, die zum Kombinieren von Parsern verwendet werden:

```
let (>>=)   t   uf  = bind t uf
let (<|>)   t   u   = orElse t u
```

Also bedeutet `>>= bind` und `<|> orElse` .

Dadurch können wir Parser prägnant kombinieren:

```
let letterOrDigit = letter <|> digit
let p = digit >>= fun v -> digit >>= fun u -> return_ (v,u)
```

Um einige erweiterte Parser-Kombinatoren zu definieren, die es uns ermöglichen, komplexere Ausdrücke zu parsen, definieren wir einige einfachere Parser-Kombinatoren:

```
// 'map' runs parser 't' and maps the result using 'm'
let map m t       = t >>= (m >> return_)
let (>>!) t m     = map m t
let (>>%) t v     = t >>! (fun _ -> v)

// 'opt' takes a parser 't' and creates a parser that always succeed but
//  if parser 't' fails the new parser will produce the value 'None'
let opt t         = (t >>! Some) <|> (return_ None)

// 'pair' runs parser 't' and 'u' and returns a pair of 't' and 'u' results
let pair t u      =
  parser {
    let! tv = t
    let! tu = u
    return tv, tu
  }
```

Wir sind bereit, `many` und `sepBy` zu definieren, die fortgeschrittener sind, wenn sie die Eingabeparser anwenden, bis sie fehlschlagen. Dann geben `many` und `sepBy` das aggregierte

Ergebnis zurück:

```
// 'many' applies parser 't' until it fails and returns all successful
//  parser results as a list
let many t =
  let ot = opt t
  let rec loop vs = ot >>= function Some v -> loop (v::vs) | None -> return_ (List.rev vs)
  loop []

// 'sepBy' applies parser 't' separated by 'sep'.
//  The values are reduced with the function 'sep' returns
let sepBy t sep     =
  let ots = opt (pair sep t)
  let rec loop v = ots >>= function Some (s, n) -> loop (s v n) | None -> return_ v
  t >>= loop
```

Einen einfachen Ausdrucksparser erstellen

Mit den von uns erstellten Tools können wir nun einen Parser für einfache Ausdrücke wie `1+2*3`

Wir beginnen von unten, indem wir einen Parser für Ganzzahlen `pint`

```
// 'pint' parses an integer
let pint =
  let f s v = 10*s + int v - int '0'
  parser {
    let! digits = many digit
    return!
      match digits with
      | [] -> fail
      | vs -> return_ (List.fold f 0 vs)
  }
```

Wir versuchen, so viele Ziffern wie möglich zu analysieren, das Ergebnis ist eine `char list` . Wenn die Liste leer ist, schlagen wir `fail` , andernfalls falten wir die Zeichen in eine Ganzzahl.

Test `pint` in FSI:

```
> run pint "123";;
val it : int option * int = (Some 123, 3)
```

Außerdem müssen wir die verschiedenen Arten von Operatoren analysieren, die zum Kombinieren von Ganzzahlen verwendet werden:

```
// operator parsers, note that the parser result is the operator function
let padd      = char '+' >>% (+)
let psubtract = char '-' >>% (-)
let pmultiply = char '*' >>% (*)
let pdivide   = char '/' >>% (/)
let pmodulus  = char '%' >>% (%)
```

FSI:

```
> run padd "+";;
```

```
val it : (int -> int -> int) option * int = (Some <fun:padd@121-1>, 1)
```

Alles zusammenbinden:

```
// 'pmullike' parsers integers separated by operators with same precedence as multiply
let pmullike  = sepBy pint (pmultiply <|> pdivide <|> pmodulus)
// 'paddlike' parsers sub expressions separated by operators with same precedence as add
let paddlike  = sepBy pmullike (padd <|> psubtract)
// 'pexpr' is the full expression
let pexpr     =
  parser {
    let! v = paddlike
    let! _ = eos      // To make sure the full string is consumed
    return v
  }
```

Alles in FSI ausführen:

```
> run pexpr "2+123*2-3";;
val it : int option * int = (Some 245, 9)
```

Fazit

Durch die Definition von `Parser<'T>` , `return_` , `bind` und sicherstellen , dass sie gehorchen den monadischen Gesetze wir eine einfache , aber leistungsfähige Monadic Parser Combinator Rahmen gebaut haben.

Monaden und Parser gehören zusammen, weil Parser in einem Parserzustand ausgeführt werden. Monaden ermöglicht es uns, Parser zu kombinieren, während der Parserzustand ausgeblendet wird. Dadurch werden Unordnung reduziert und die Komposierbarkeit verbessert.

Das von uns erstellte Framework ist langsam und erzeugt keine Fehlermeldungen, um den Code übersichtlich zu halten. FParsec bietet sowohl eine akzeptable Leistung als auch hervorragende Fehlermeldungen.

Ein Beispiel allein kann Monaden jedoch nicht verstehen. Man muss Monaden üben.

Hier einige Beispiele für Monaden, die Sie implementieren können, um Ihr Verständnis zu erreichen:

1. State Monad - Ermöglicht das implizite Tragen eines versteckten Umgebungsstatus
2. Tracer Monad - Ermöglicht das implizite Tragen des Trace-Status. Eine Variante von State Monad
3. Turtle Monad - Eine Monade zum Erstellen von Turtle (Logos) -Programmen. Eine Variante von State Monad
4. Fortsetzung Monade - Eine Coroutine-Monade. Ein Beispiel dafür ist `async` in F #

Das Beste, um zu lernen, wäre, eine Anwendung für Monaden in einer Domäne zu finden, mit der Sie sich wohl fühlen. Für mich waren das Parser.

Vollständiger Quellcode:

```
// A Parser<'T> is a function that takes a string and position
//  and returns an optionally parsed value and a position
//  A parsed value means the position points to the character following the parsed value
//  No parsed value indicates a parse failure at the position
type Parser<'T> = Parser of (string*int -> 'T option*int)

// Runs a parser 't' on the input string 's'
let run t s =
  let (Parser tps) = t
  tps (s, 0)

// Different ways to create parser result
let succeedWith v p = Some v, p
let failAt        p = None  , p

// The 'satisfy' parser succeeds if the character at the current position
//  passes the 'sat' function
let satisfy sat : Parser<char> = Parser <| fun (s, p) ->
  if p < s.Length && sat s.[p] then succeedWith s.[p] (p + 1)
  else failAt p

// 'eos' succeeds if the position is beyond the last character.
//  Useful when testing if the parser have consumed the full string
let eos : Parser<unit> = Parser <| fun (s, p) ->
  if p < s.Length then failAt p
  else succeedWith () p

let anyChar       = satisfy (fun _ -> true)
let char ch       = satisfy ((=) ch)
let digit         = satisfy System.Char.IsDigit
let letter        = satisfy System.Char.IsLetter

// 'fail' is a parser that always fails
let fail<'T>      = Parser <| fun (s, p) -> failAt p
// 'return_' is a parser that always succeed with value 'v'
let return_ v     = Parser <| fun (s, p) -> succeedWith v p

// 'bind' let us combine two parser into a more complex parser
let bind t uf     = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    -> None, tp
  | Some tv ->
    let u = uf tv
    let (Parser ups) = u
    ups (s, tp)

type ParserBuilder() =
  member x.Bind       (t, uf) = bind      t   uf
  member x.Return     v       = return_   v
  member x.ReturnFrom t       = t

// 'parser' enables us to combine parsers using 'parser { ... }' syntax
let parser = ParserBuilder()

// 'orElse' creates a parser that runs parser 't' first, if that is successful
//  the result is returned otherwise the result of parser 'u' is returned
let orElse t u    = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
```

```
  match tov with
  | None    ->
    let (Parser ups) = u
    ups (s, p)
  | Some tv -> succeedWith tv tp

let (>>=) t uf    = bind t uf
let (<|>) t u     = orElse t u

// 'map' runs parser 't' and maps the result using 'm'
let map m t       = t >>= (m >> return_)
let (>>!) t m     = map m t
let (>>%) t v     = t >>! (fun _ -> v)

// 'opt' takes a parser 't' and creates a parser that always succeed but
//  if parser 't' fails the new parser will produce the value 'None'
let opt t         = (t >>! Some) <|> (return_ None)

// 'pair' runs parser 't' and 'u' and returns a pair of 't' and 'u' results
let pair t u      =
  parser {
    let! tv = t
    let! tu = u
    return tv, tu
  }

// 'many' applies parser 't' until it fails and returns all successful
//  parser results as a list
let many t =
  let ot = opt t
  let rec loop vs = ot >>= function Some v -> loop (v::vs) | None -> return_ (List.rev vs)
  loop []

// 'sepBy' applies parser 't' separated by 'sep'.
//  The values are reduced with the function 'sep' returns
let sepBy t sep     =
  let ots = opt (pair sep t)
  let rec loop v = ots >>= function Some (s, n) -> loop (s v n) | None -> return_ v
  t >>= loop

// A simplistic integer expression parser

// 'pint' parses an integer
let pint =
  let f s v = 10*s + int v - int '0'
  parser {
    let! digits = many digit
    return!
      match digits with
      | [] -> fail
      | vs -> return_ (List.fold f 0 vs)
  }

// operator parsers, note that the parser result is the operator function
let padd      = char '+' >>% (+)
let psubtract = char '-' >>% (-)
let pmultiply = char '*' >>% (*)
let pdivide   = char '/' >>% (/)
let pmodulus  = char '%' >>% (%)

// 'pmullike' parsers integers separated by operators with same precedence as multiply
```

```
let pmullike  = sepBy pint (pmultiply <|> pdivide <|> pmodulus)
// 'paddlike' parsers sub expressions separated by operators with same precedence as add
let paddlike  = sepBy pmullike (padd <|> psubtract)
// 'pexpr' is the full expression
let pexpr     =
  parser {
    let! v = paddlike
    let! _ = eos      // To make sure the full string is consumed
    return v
  }
```

Berechnungsausdrücke bieten eine alternative Syntax zur Verkettung von Monaden

Mit Monaden verwandt sind `F#` -Berechnungsausdrücke (`CE`). Ein Programmierer implementiert normalerweise ein `CE` , um einen alternativen Ansatz zur Verkettung von Monaden bereitzustellen, dh stattdessen:

```
let v = m >>= fun x -> n >>= fun y -> return_ (x, y)
```

Sie können dies schreiben:

```
let v = ce {
    let! x = m
    let! y = n
    return x, y
  }
```

Beide Stile sind gleichwertig und es liegt an den von den Entwicklern bevorzugten, welchen Sie wählen.

Um zu zeigen, wie ein `CE` implementiert wird, stellen Sie sich vor, dass alle Spuren eine Korrelations-ID enthalten. Diese Korrelations-ID hilft bei der Korrelation von Spuren, die zu demselben Anruf gehören. Dies ist sehr nützlich, wenn Protokolldateien mit Ablaufverfolgungen von gleichzeitigen Aufrufen vorhanden sind.

Das Problem ist, dass es mühsam ist, die Korrelations-ID als Argument für alle Funktionen anzugeben. Da Monaden den impliziten Zustand zulassen, definieren wir eine Log-Monade, um den Log-Kontext (dh die Korrelations-ID) auszublenden.

Wir definieren zunächst einen Protokollkontext und den Typ einer Funktion, die mit dem Protokollkontext verfolgt:

```
type Context =
  {
    CorrelationId : Guid
  }
  static member New () : Context = { CorrelationId = Guid.NewGuid () }

type Function<'T> = Context -> 'T

// Runs a Function<'T> with a new log context
```

```
let run t = t (Context.New ())
```

Wir definieren auch zwei Trace-Funktionen, die mit der Korrelations-ID aus dem Protokollkontext protokolliert werden:

```
let trace v   : Function<_> = fun ctx -> printfn "CorrelationId: %A - %A" ctx.CorrelationId v
let tracef fmt              = kprintf trace fmt
```

`trace` ist eine `Function<unit>` was bedeutet, dass beim Aufruf ein Protokollkontext übergeben wird. Aus dem Log-Kontext nehmen wir die Korrelations-ID auf und verfolgen sie zusammen mit `v`

Zusätzlich definieren wir `bind` und `return_` und da sie den Monad-Gesetzen folgen, bildet dies unsere Log Monad.

```
let bind t uf : Function<_> = fun ctx ->
  let tv = t ctx  // Invoke t with the log context
  let u  = uf tv  // Create u function using result of t
  u ctx           // Invoke u with the log context

// >>= is the common infix operator for bind
let inline (>>=) (t, uf) = bind t uf

let return_ v : Function<_> = fun ctx -> v
```

Schließlich definieren wir `LogBuilder` , mit dem wir die `CE` Syntax verwenden können, um `Log` `LogBuilder` .

```
type LogBuilder() =
  member x.Bind   (t, uf) = bind t uf
  member x.Return v       = return_ v

// This enables us to write function like: let f = log { ... }
let log = Log.LogBuilder ()
```

Wir können jetzt unsere Funktionen definieren, die den impliziten Protokollkontext haben sollten:

```
let f x y =
  log {
    do! Log.tracef "f: called with: x = %d, y = %d" x y
    return x + y
  }

let g =
  log {
    do! Log.trace "g: starting..."
    let! v = f 1 2
    do! Log.tracef "g: f produced %d" v
    return v
  }
```

Wir führen g aus mit:

```
printfn "g produced %A" (Log.run g)
```

Welche drucke:

```
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "g: starting..."
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "f: called with: x = 1, y = 2"
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "g: f produced 3"
g produced 3
```

Beachten Sie, dass die CorrelationId implizit von `run` nach `g` bis `f` , wodurch wir die Protokolleinträge während der Fehlersuche korrelieren können.

`CE` hat viel mehr Funktionen, aber dies sollte Ihnen helfen, Ihre eigenen `CE` definieren.

Vollständiger Code:

```
module Log =
  open System
  open FSharp.Core.Printf

  type Context =
    {
      CorrelationId : Guid
    }
    static member New () : Context = { CorrelationId = Guid.NewGuid () }

  type Function<'T> = Context -> 'T

  // Runs a Function<'T> with a new log context
  let run t = t (Context.New ())

  let trace v   : Function<_> = fun ctx -> printfn "CorrelationId: %A - %A" ctx.CorrelationId
v
  let tracef fmt              = kprintf trace fmt

  let bind t uf : Function<_> = fun ctx ->
    let tv = t ctx  // Invoke t with the log context
    let u  = uf tv  // Create u function using result of t
    u ctx           // Invoke u with the log context

  // >>= is the common infix operator for bind
  let inline (>>=) (t, uf) = bind t uf

  let return_ v : Function<_> = fun ctx -> v

  type LogBuilder() =
    member x.Bind   (t, uf) = bind t uf
    member x.Return v       = return_ v

// This enables us to write function like: let f = log { ... }
let log = Log.LogBuilder ()

let f x y =
  log {
    do! Log.tracef "f: called with: x = %d, y = %d" x y
    return x + y
  }

let g =
  log {
```

```
    do! Log.trace "g: starting..."
    let! v = f 1 2
    do! Log.tracef "g: f produced %d" v
    return v
  }

[<EntryPoint>]
let main argv =
  printfn "g produced %A" (Log.run g)
  0
```

Monaden online lesen: https://riptutorial.com/de/fsharp/topic/3320/monaden

Kapitel 21: Musterabgleich

Bemerkungen

Pattern Matching ist eine leistungsstarke Funktion in vielen funktionalen Sprachen, da Verzweigungen häufig im Vergleich zur Verwendung mehrerer `if` / `else if` / `else` Anweisungen sehr präzise gehandhabt werden können. Wenn Sie jedoch genügend Optionen und "Wann" - Wächter haben , kann Pattern Matching auf einen Blick auch verbos und schwer verständlich werden.

In diesem Fall können die Active Patterns von F # eine hervorragende Möglichkeit sein, der Abgleichlogik sinnvolle Namen zu geben, was den Code vereinfacht und die Wiederverwendung ermöglicht.

Examples

Übereinstimmende Optionen

Musterabgleich kann hilfreich sein, um mit Optionen umzugehen:

```
let result = Some("Hello World")
match result with
| Some(message) -> printfn message
| None -> printfn "Not feeling talkative huh?"
```

Mustervergleichsüberprüfungen der gesamten Domäne werden abgedeckt

```
let x = true
match x with
| true -> printfn "x is true"
```

gibt eine Warnung aus

> C: \ Programme (x86) \ Microsoft VS Code \ Untitled-1 (2,7): Warnung FS0025: Unvollständiges Muster stimmt mit diesem Ausdruck überein. Zum Beispiel kann der Wert 'false' auf einen Fall hinweisen, der nicht durch die Muster abgedeckt ist.

Dies liegt daran, dass nicht alle möglichen bool-Werte abgedeckt wurden.

bools können explizit aufgelistet werden, aber es ist schwieriger, sie aufzulisten

```
let x = 5
match x with
| 1 -> printfn "x is 1"
| 2 -> printfn "x is 2"
| _ -> printfn "x is something else"
```

Hier verwenden wir das Sonderzeichen _ . Das _ stimmt mit allen anderen möglichen Fällen überein.

Der _ kann Sie in Schwierigkeiten bringen

Betrachten wir einen Typ, den wir selbst erstellen, sieht es so aus

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
```

Wir könnten ein Match mit Expession schreiben, das so aussieht

```
match sobriety with
| Sober -> printfn "drive home"
| _ -> printfn "call an uber"
```

Der obige Code ist sinnvoll. Wenn Sie nicht nüchtern sind, gehen wir davon aus, dass Sie eine Uber anrufen sollten, damit wir das _ angeben

Wir überarbeiten unseren Code später darauf

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
    | Unconscious
```

Der F # -Compiler sollte uns eine Warnung geben und uns auffordern, unseren Übereinstimmungsausdruck zu ändern, damit die Person einen Arzt aufsucht. Stattdessen behandelt der Übereinstimmungsausdruck die unbewusste Person stumm, als ob sie nur angetrunken wäre. Der Punkt ist, dass Sie sich dazu entscheiden sollten, Fälle explizit aufzulisten, wenn möglich, um logische Fehler zu vermeiden.

Die Fälle werden von oben nach unten ausgewertet und die erste Übereinstimmung wird verwendet

Falsche Verwendung:

Im folgenden Snippet wird das letzte Match niemals verwendet:

```
let x = 4
```

```
match x with
| 1 -> printfn "x is 1"
| _ -> printfn "x is anything that wasn't listed above"
| 4 -> printfn "x is 4"
```

druckt

> x ist alles, was oben nicht aufgeführt wurde

Richtige Benutzung:

Hier treffen sowohl `x = 1` als auch `x = 4` auf ihre spezifischen Fälle, während alles andere auf den Standardfall _ fällt:

```
let x = 4
match x with
| 1 -> printfn "x is 1"
| 4 -> printfn "x is 4"
| _ -> printfn "x is anything that wasn't listed above"
```

druckt

> x ist 4

Bei Wachen können Sie beliebige Bedingungen hinzufügen

```
type Person = {
    Age : int
    PassedDriversTest : bool }

let someone = { Age = 19; PassedDriversTest = true }

match someone.PassedDriversTest with
| true when someone.Age >= 16 -> printfn "congrats"
| true -> printfn "wait until you are 16"
| false -> printfn "you need to pass the test"
```

Musterabgleich online lesen: https://riptutorial.com/de/fsharp/topic/1335/musterabgleich

Kapitel 22: Operatoren

Examples

Wie man Werte und Funktionen mit gemeinsamen Operatoren zusammenstellt

In der objektorientierten Programmierung besteht die übliche Aufgabe darin, Objekte (Werte) zusammenzustellen. In der funktionalen Programmierung ist es ebenso üblich, Werte und Funktionen zusammenzustellen.

Mit Hilfe von Operatoren wie wir werden Werte von unserer Erfahrung aus anderen Programmiersprachen zu komponieren `+` , `-` , `*` , `/` und so weiter.

Wertzusammensetzung

```
let x = 1 + 2 + 3 * 2
```

Da die Funktionsprogrammierung sowohl Funktionen als auch Werte enthält, ist es nicht überraschend, dass es für die Funktionszusammenstellung übliche Operatoren wie `>>` , `<<` , `|>` und `<|` .

Funktionszusammenstellung

```
// val f : int -> int
let f v   = v + 1
// val g : int -> int
let g v   = v * 2

// Different ways to compose f and g
// val h : int -> int
let h1 v  = g (f v)
let h2 v  = v |> f |> g   // Forward piping of 'v'
let h3 v  = g <| (f <| v) // Reverse piping of 'v' (because <| has left associcativity we need
())
let h4    = f >> g        // Forward functional composition
let h5    = g << f        // Reverse functional composition (closer to math notation of 'g o
f')
```

In `F#` Vorwärtsleitung vor der Rückwärtsleitung bevorzugt, weil:

1. Typinferenz fließt (im Allgemeinen) von links nach rechts, sodass Werte und Funktionen auch von links nach rechts fließen können
2. Weil `<|` und `<<` sollte rechte Assoziativität haben, aber in `F#` bleiben sie assoziativ, was uns zwingt, einzufügen ()
3. Das Mischen von vorwärts und rückwärts verlaufenden Rohren funktioniert im Allgemeinen nicht, da sie die gleiche Priorität haben.

Monadenzusammensetzung

Da Monaden (wie `Option<'T>` oder `List<'T>`) häufig in der Funktionsprogrammierung verwendet werden, gibt es auch übliche, aber weniger bekannte Operatoren, die Funktionen erstellen, die mit Monaden arbeiten, wie `>>=` , `>=>` , `<|>` und `<*>` .

```
let (>>=) t uf  = Option.bind uf t
let (>=>) tf uf = fun v -> tf v >>= uf
// val oinc   : int -> int option
let oinc   v    = Some (v + 1)    // Increment v
// val ofloat : int -> float option
let ofloat v    = Some (float v)  // Map v to float

// Different ways to compose functions working with Option Monad
// val m : int option -> float option
let m1 v  = Option.bind (fun v -> Some (float (v + 1))) v
let m2 v  = v |> Option.bind oinc |> Option.bind ofloat
let m3 v  = v >>= oinc >>= ofloat
let m4    = oinc >=> ofloat

// Other common operators are <|> (orElse) and <*> (andAlso)

// If 't' has Some value then return t otherwise return u
let (<|>) t u =
  match t with
  | Some _  -> t
  | None    -> u

// If 't' and 'u' has Some values then return Some (tv*uv) otherwise return None
let (<*>) t u =
  match t, u with
  | Some tv, Some tu  -> Some (tv, tu)
  | _                 -> None

// val pickOne : 'a option -> 'a option -> 'a option
let pickOne t u v = t <|> u <|> v

// val combine : 'a option -> 'b option  -> 'c option -> (('a*'b)*'c) option
let combine t u v = t <*> u <*> v
```

Fazit

Für neue funktionale Programmierer mag die Funktionszusammenstellung mit Operatoren undurchsichtig und unklar sein, aber das liegt daran, dass die Bedeutung dieser Operatoren nicht so allgemein als Operatoren bekannt ist, die mit Werten arbeiten. Mit etwas Training mit `|>` wird `>>` `>>=` und `>=>` so selbstverständlich wie mit `+` , `-` , `*` und `/` .

Latebinding in F # mit? Operator

In einer statisch typisierten Sprache wie `F#` wir mit Typen, die zur Kompilierzeit bekannt sind. Wir konsumieren externe Datenquellen typsicher mit Typanbietern.

Gelegentlich muss jedoch die späte Bindung verwendet werden (z. B. `dynamic` in `C#`). Zum Beispiel beim Arbeiten mit `JSON` Dokumenten, die kein klar definiertes Schema haben.

Um die Arbeit mit Late Binding zu vereinfachen, unterstützt `F#` dynamische Suchoperatoren `?` und `?<-` .

Beispiel:

```
// (?) allows us to lookup values in a map like this: map?MyKey
let inline (?)   m k   = Map.tryFind k m
// (?<-) allows us to update values in a map like this: map?MyKey <- 123
let inline (?<-) m k v = Map.add k v m

let getAndUpdate (map : Map<string, int>) : int option*Map<string, int> =
  let i = map?Hello       // Equivalent to map |> Map.tryFind "Hello"
  let m = map?Hello <- 3  // Equivalent to map |> Map.add "Hello" 3
  i, m
```

Es stellt sich heraus, dass die `F#` -Unterstützung für spätes Binden einfach und dennoch flexibel ist.

Operatoren online lesen: https://riptutorial.com/de/fsharp/topic/4641/operatoren

Kapitel 23: Optionstypen

Examples

Definition der Option

Eine `Option` ist eine diskriminierte Gewerkschaft mit zwei Fällen: `None` oder `Some` .

```
type Option<'T> = Some of 'T | None
```

Verwenden Sie die Option <'T> über Nullwerte

In funktionalen Programmiersprachen wie `F# null` als potenziell schädlich und als schlecht (nicht idiomatisch) angesehen.

Betrachten Sie diesen `C#` -Code:

```
string x = SomeFunction ();
int    l = x.Length;
```

`x.Length` wird geworfen, wenn `x null` , fügen Sie den Schutz hinzu:

```
string x = SomeFunction ();
int    l = x != null ? x.Length : 0;
```

Oder:

```
string x = SomeFunction () ?? "";
int    l = x.Length;
```

Oder:

```
string x = SomeFunction ();
int    l = x?.Length;
```

In idiomatic werden keine `F# null` verwendet, daher sieht unser Code folgendermaßen aus:

```
let x = SomeFunction ()
let l = x.Length
```

Manchmal müssen jedoch leere oder ungültige Werte dargestellt werden. Dann können wir `Option<'T>` :

```
let SomeFunction () : string option = ...
```

`SomeFunction` entweder `Some string` oder `None` . Wir extrahieren den `string` Wert mithilfe von Pattern-

Matching

```
let v =
  match SomeFunction () with
  | Some x  -> x.Length
  | None    -> 0
```

Der Grund, warum dieser Code weniger anfällig ist als:

```
string x = SomeFunction ();
int    l = x.Length;
```

Ist, weil wir `Length` mit einer `string option` aufrufen können. Wir müssen den `string` Wert mithilfe von Pattern-Matching extrahieren. Dadurch wird sichergestellt, dass der `string` Wert sicher verwendet werden kann.

Das Optionsmodul ermöglicht die eisenbahnorientierte Programmierung

Fehlerbehandlung ist wichtig, kann jedoch einen eleganten Algorithmus zum Durcheinander bringen. Eisenbahnorientierte Programmierung (`ROP`) wird verwendet, um die Fehlerbehandlung elegant und komponierbar zu gestalten.

Betrachten Sie die einfache Funktion `f` :

```
let tryParse s =
  let b, v = System.Int32.TryParse s
  if b then Some v else None

let f (g : string option) : float option =
  match g with
  | None    -> None
  | Some s  ->
    match tryParse s with           // Parses string to int
    | None              -> None
    | Some v when v < 0 -> None     // Checks that int is greater than 0
    | Some v -> v |> float |> Some  // Maps int to float
```

Der Zweck von `f` ist die Eingabe zu analysieren `string` - Wert (wenn es `Some`) in einen `int` . Wenn das `int` größer als `0` , werfen wir es in einen `float` . In allen anderen Fällen retten wir mit `None` .

Eine extrem einfache Funktion, die verschachtelte `match` verringert jedoch die Lesbarkeit erheblich.

`ROP` stellt fest, dass wir zwei Arten von Ausführungspfaden in unserem Programm haben

1. Glücklicher Weg - Will irgendwann `Some` Wert berechnen
2. Fehlerpfad - Alle anderen Pfade erzeugen `None`

Da die Fehlerpfade häufiger sind, neigen sie dazu, den Code zu übernehmen. Wir möchten, dass der Happy-Path-Code der sichtbarste Codepfad ist.

Eine äquivalente Funktion `g` die `ROP` könnte folgendermaßen aussehen:

```
let g (v : string option) : float option =
  v
  |> Option.bind    tryParse  // Parses string to int
  |> Option.filter  ((<) 0)   // Checks that int is greater than 0
  |> Option.map     float     // Maps int to float
```

Es sieht sehr danach aus, wie wir Listen und Sequenzen in `F#` .

Man kann eine `Option<'T>` wie eine `List<'T>` , die nur `0` oder `1` Elemente enthalten kann, wobei `Option.bind` sich wie `List.pick` verhält (konzeptionell ist `Option.bind` besser auf `List.collect` `List.pick` möglicherweise jedoch `List.pick` leichter zu verstehen).

`bind` , `filter` und `map` behandelt die Fehlerpfade und `g` enthält nur den Code für den glücklichen Pfad.

Alle Funktionen, die `Option<_>` direkt akzeptieren und `Option<_>` direkt mit `|>` und `>>` .

`ROP` erhöht daher die Lesbarkeit und Komposierbarkeit.

Verwendung von Optionstypen aus C

Es ist keine gute Idee, Optionstypen mit C # -Code zu versehen, da C # keine Möglichkeit hat, sie zu behandeln. Sie haben die Möglichkeit, entweder `FSharp.Core` als Abhängigkeit in Ihrem C # -Projekt einzuführen (was Sie tun müssten, wenn Sie eine F # -Bibliothek verwenden, die *nicht* für die Interop mit C # vorgesehen ist) oder die Werte von `None` in `null` ändern.

Pre-F # 4.0

Dazu erstellen Sie eine eigene Konvertierungsfunktion:

```
let OptionToObject opt =
    match opt with
    | Some x -> x
    | None -> null
```

Für `System.Nullable` Sie auf Boxen oder `System.Nullable` .

```
let OptionToNullable x =
    match x with
    | Some i -> System.Nullable i
    | None -> System.Nullable ()
```

F # 4.0

In F # 4.0 wurden die Funktionen `ofObj` , `toObj` , `ofNullable` und `toNullable` in das `Option` . In F # interactive können sie wie folgt verwendet werden:

```
let l1 = [ Some 1 ; None ; Some 2]
let l2 = l1 |> List.map Option.toNullable;;

// val l1 : int option list = [Some 1; null; Some 2]
// val l2 : System.Nullable<int> list = [1; null; 2]

let l3 = l2 |> List.map Option.ofNullable;;
// val l3 : int option list = [Some 1; null; Some 2]

// Equality
l1 = l2 // fails to compile: different types
l1 = l3 // true
```

Beachten Sie, dass `None` intern zu `null` kompiliert wird. Für F # ist dies jedoch `None` .

```
let lA = [Some "a"; None; Some "b"]
let lB = lA |> List.map Option.toObj

// val lA : string option list = [Some "a"; null; Some "b"]
// val lB : string list = ["a"; null; "b"]

let lC = lB |> List.map Option.ofObj
// val lC : string option list = [Some "a"; null; Some "b"]

// Equality
lA = lB // fails to compile: different types
lA = lC // true
```

Optionstypen online lesen: https://riptutorial.com/de/fsharp/topic/3175/optionstypen

Kapitel 24: Portierung von C # nach F

Examples

POCOs

Einige der einfachsten Klassen sind POCOs.

```
// C#
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }
}
```

In F # 3.0 wurden Auto-Eigenschaften eingeführt, die den C # Auto-Eigenschaften ähneln.

```
// F#
type Person() =
    member val FirstName = "" with get, set
    member val LastName = "" with get, set
    member val BirthDay = System.DateTime.Today with get, set
```

Die Erstellung einer Instanz von beiden ist ähnlich,

```
// C#
var person = new Person { FirstName = "Bob", LastName = "Smith", Birthday = DateTime.Today };
// F#
let person = new Person(FirstName = "Bob", LastName = "Smith")
```

Wenn Sie unveränderliche Werte verwenden können, ist ein Datensatztyp viel idiomatischer F #.

```
type Person = {
    FirstName:string;
    LastName:string;
    Birthday:System.DateTime
}
```

Und dieser Datensatz kann erstellt werden:

```
let person = { FirstName = "Bob"; LastName = "Smith"; Birthday = System.DateTime.Today }
```

Datensätze können auch basierend auf anderen Datensätzen erstellt werden, indem der vorhandene Datensatz angegeben und `with` einer Liste von zu überschreibenden Feldern hinzugefügt wird:

```
let formal = { person with FirstName = "Robert" }
```

Klasse Eine Schnittstelle implementieren

Klassen implementieren eine Schnittstelle, um den Vertrag der Schnittstelle zu erfüllen. Zum Beispiel kann eine C # -Klasse `IDisposable` implementieren,

```
public class Resource : IDisposable
{
    private MustBeDisposed internalResource;

    public Resource()
    {
        internalResource = new MustBeDisposed();
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing){
        if (disposing){
            if (resource != null) internalResource.Dispose();
        }
    }
}
```

Um eine Schnittstelle in F # zu implementieren, verwenden Sie die `interface` in der Typdefinition.

```
type Resource() =
    let internalResource = new MustBeDisposed()

    interface IDisposable with
        member this.Dispose(): unit =
            this.Dispose(true)
            GC.SuppressFinalize(this)

    member __.Dispose disposing =
        match disposing with
        | true  -> if (not << isNull) internalResource then internalResource.Dispose()
        | false -> ()
```

Portierung von C # nach F # online lesen: https://riptutorial.com/de/fsharp/topic/6828/portierung-von-c-sharp-nach-f-sharp

Kapitel 25: Postfachprozessor

Bemerkungen

`MailboxProcessor` verwaltet eine interne Nachrichtenwarteschlange, in der mehrere Produzenten Nachrichten mit verschiedenen `Post` Methodenvarianten `MailboxProcessor` können. Diese Nachrichten werden dann von einem einzelnen Consumer abgerufen und verarbeitet (sofern Sie ihn nicht anderweitig implementieren), und zwar mithilfe der Varianten " `Retrieve` und " `Scan` . Standardmäßig ist das Erstellen und Verwenden der Nachrichten Thread-sicher.

Standardmäßig ist keine Fehlerbehandlung vorgesehen. Wenn eine nicht erfasste Ausnahme im Rumpf des Prozessors ausgelöst wird, wird die Rumpffunktion beendet, alle Nachrichten in der Warteschlange gehen verloren, es können keine weiteren Nachrichten mehr gesendet werden, und der Antwortkanal (falls verfügbar) erhält eine Ausnahme anstelle einer Antwort. Sie müssen die Fehlerbehandlung selbst vornehmen, falls dieses Verhalten nicht zu Ihrem Anwendungsfall passt.

Examples

Grundlegende Hallo Welt

Lassen Sie uns zunächst eine einfache "Hallo Welt!" `MailboxProcessor` der einen Nachrichtentyp verarbeitet und Begrüßungen `MailboxProcessor` .

Sie benötigen den Nachrichtentyp. Es kann alles sein, aber diskriminierte Vereinigungen sind eine natürliche Wahl, da sie alle möglichen Fälle an einem Ort auflisten und Sie beim Zuordnen von Mustern problemlos den Musterabgleich verwenden können.

```
// In this example there is only a single case, it could technically be just a string
type GreeterMessage = SayHelloTo of string
```

Definieren Sie nun den Prozessor selbst. Dies kann mit `MailboxProcessor<'message>.Start` Statische Methode `MailboxProcessor<'message>.Start` , die einen gestarteten Prozessor zurückgibt, der bereit ist, seinen Job auszuführen. Sie können auch den Konstruktor verwenden, müssen jedoch sicherstellen, dass der Prozessor später gestartet wird.

```
let processor = MailboxProcessor<GreeterMessage>.Start(fun inbox ->
    let rec innerLoop () = async {
        // This way you retrieve message from the mailbox queue
        // or await them in case the queue empty.
        // You can think of the `inbox` parameter as a reference to self.
        let! message = inbox.Receive()
        // Now you can process the retrieved message.
        match message with
        | SayHelloTo name ->
            printfn "Hi, %s! This is mailbox processor's inner loop!" name
        // After that's done, don't forget to recurse so you can process the next messages!
```

```
        innerLoop()
    }
    innerLoop ())
```

Der Parameter zu `Start` ist eine Funktion , die einen Verweis auf das nimmt `MailboxProcessor` selbst (die noch nicht existiert , wie Sie es nur schaffen, wird aber verfügbar sein , sobald die Funktion ausführt). Dadurch haben Sie Zugriff auf die verschiedenen `Receive` und `Scan` Methoden, um auf die Nachrichten aus dem Postfach zuzugreifen. Innerhalb dieser Funktion können Sie die Verarbeitung ausführen, die Sie benötigen. Normalerweise wird jedoch eine Endlosschleife verwendet, die die Nachrichten nacheinander liest und sich nach jeder einzelnen Nachricht aufruft.

Jetzt ist der Prozessor fertig, aber es geht nichts! Warum? Sie müssen eine Nachricht senden, um sie zu bearbeiten. Dies geschieht mit den `Post` Methodenvarianten - verwenden wir die einfachsten, feuervergessenden Methoden.

```
processor.Post(SayHelloTo "Alice")
```

Dadurch wird eine Nachricht in die interne Warteschlange des `processor` , das Postfach, eingefügt und sofort zurückgegeben, sodass der aufrufende Code fortgesetzt werden kann. Sobald der Prozessor die Nachricht abgerufen hat, wird er sie verarbeiten. Dies geschieht jedoch asynchron zum Posten der Nachricht und wird höchstwahrscheinlich in einem separaten Thread durchgeführt.

Sehr bald danach sollte die Meldung `"Hi, Alice! This is mailbox processor's inner loop!"` zur Ausgabe gedruckt und Sie sind bereit für kompliziertere Proben.

Mutable State Management

Postfachprozessoren können verwendet werden, um den veränderbaren Status auf transparente und Thread-sichere Weise zu verwalten. Lass uns einen einfachen Zähler bauen.

```
// Increment or decrement by one.
type CounterMessage =
    | Increment
    | Decrement

let createProcessor initialState =
    MailboxProcessor<CounterMessage>.Start(fun inbox ->
        // You can represent the processor's internal mutable state
        // as an immutable parameter to the inner loop function
        let rec innerLoop state = async {
            printfn "Waiting for message, the current state is: %i" state
            let! message = inbox.Receive()
            // In each call you use the current state to produce a new
            // value, which will be passed to the next call, so that
            // next message sees only the new value as its local state
            match message with
            | Increment ->
                let state' = state + 1
                printfn "Counter incremented, the new state is: %i" state'
                innerLoop state'
            | Decrement ->
```

```
                let state' = state - 1
                printfn "Counter decremented, the new state is: %i" state'
                innerLoop state'
        }
        // We pass the initialState to the first call to innerLoop
        innerLoop initialState)

// Let's pick an initial value and create the processor
let processor = createProcessor 10
```

Lassen Sie uns nun einige Operationen generieren

```
processor.Post(Increment)
processor.Post(Increment)
processor.Post(Decrement)
processor.Post(Increment)
```

Das folgende Protokoll wird angezeigt

```
Waiting for message, the current state is: 10
Counter incremented, the new state is: 11
Waiting for message, the current state is: 11
Counter incremented, the new state is: 12
Waiting for message, the current state is: 12
Counter decremented, the new state is: 11
Waiting for message, the current state is: 11
Counter incremented, the new state is: 12
Waiting for message, the current state is: 12
```

Parallelität

Da der Postfachprozessor die Nachrichten einzeln verarbeitet und es keine Verschachtelung gibt, können Sie die Nachrichten auch aus mehreren Threads erstellen und die typischen Probleme verlorener oder duplizierter Vorgänge nicht sehen. Es ist nicht möglich, dass eine Nachricht den alten Status anderer Nachrichten verwendet, es sei denn, Sie implementieren den Prozessor ausdrücklich.

```
let processor = createProcessor 0

[ async { processor.Post(Increment) }
  async { processor.Post(Increment) }
  async { processor.Post(Decrement) }
  async { processor.Post(Decrement) } ]
|> Async.Parallel
|> Async.RunSynchronously
```

Alle Nachrichten werden aus verschiedenen Threads gepostet. Die Reihenfolge, in der Nachrichten an das Postfach gesendet werden, ist nicht deterministisch. Daher ist die Reihenfolge der Verarbeitung nicht deterministisch. Da jedoch die Gesamtanzahl der Inkremente und Dekremente ausgeglichen ist, wird der Endstatus unabhängig von der Reihenfolge 0 angezeigt und von welchen Threads die Nachrichten gesendet wurden.

Wirklich veränderlicher Zustand

Im vorherigen Beispiel haben wir nur den veränderlichen Status durch Übergeben des rekursiven Schleifenparameters simuliert. Der Postfachprozessor verfügt jedoch über alle diese Eigenschaften, selbst für einen wirklich veränderbaren Status. Dies ist wichtig, wenn Sie einen großen Zustand beibehalten und die Unveränderlichkeit aus Leistungsgründen nicht praktikabel ist.

Wir können unseren Gegencode für die folgende Implementierung umschreiben

```
let createProcessor initialState =
    MailboxProcessor<CounterMessage>.Start(fun inbox ->
        // In this case we represent the state as a mutable binding
        // local to this function. innerLoop will close over it and
        // change its value in each iteration instead of passing it around
        let mutable state = initialState

        let rec innerLoop () = async {
            printfn "Waiting for message, the current state is: %i" state
            let! message = inbox.Receive()
            match message with
            | Increment ->
                let state <- state + 1
                printfn "Counter incremented, the new state is: %i" state'
                innerLoop ()
            | Decrement ->
                let state <- state - 1
                printfn "Counter decremented, the new state is: %i" state'
                innerLoop ()
        }
        innerLoop ())
```

Auch wenn dies definitiv nicht Thread-sicher wäre, wenn der Zählerstand direkt von mehreren Threads geändert wurde, können Sie anhand der parallelen Nachricht Posts aus dem vorherigen Abschnitt sehen, dass der Postfachprozessor die Nachrichten ohne Interleaving nacheinander verarbeitet aktuellster Wert.

Rückgabewerte

Sie können einen Wert für jede verarbeitete Nachricht asynchron zurückgeben, wenn Sie einen `AsyncReplyChannel<'a>` als Teil der Nachricht senden.

```
type MessageWithResponse = Message of InputData * AsyncReplyChannel<OutputData>
```

Dann kann der Mailbox-Prozessor diesen Kanal bei der Verarbeitung der Nachricht verwenden, um einen Wert an den Anrufer zurückzusenden.

```
let! message = inbox.Receive()
match message with
| MessageWithResponse (data, r) ->
    // ...process the data
    let output = ...
```

```
    r.Reply(output)
```

`AsyncReplyChannel<'a>` jetzt eine Nachricht zu erstellen, benötigen Sie den `AsyncReplyChannel<'a>` Was ist das und wie erstellen Sie eine Arbeitsinstanz? Der beste Weg ist, MailboxProcessor es für Sie zur Verfügung stellen zu lassen und die Antwort an ein häufigeres `Async<'a>` extrahieren. Dies kann beispielsweise mit der `PostAndAsynReply` Methode erfolgen, bei der Sie nicht die vollständige Nachricht veröffentlichen, sondern stattdessen eine Funktion vom Typ (in unserem Fall) `AsyncReplyChannel<OutputData> -> MessageWithResponse` :

```
let! output = processor.PostAndAsyncReply(r -> MessageWithResponse(input, r))
```

Dadurch wird die Nachricht in eine Warteschlange gestellt und auf die Antwort gewartet, die eintreffen wird, wenn der Prozessor diese Nachricht erhält und über den Kanal antwortet.

Es gibt auch eine synchrone Variante `PostAndReply` die den aufrufenden Thread blockiert, bis der Prozessor antwortet.

Out-of-Order-Nachrichtenverarbeitung

Sie können mit `Scan` oder `TryScan` Methoden für bestimmte Nachrichten in der Warteschlange suchen und verarbeiten sie , unabhängig davon , wie viele Nachrichten sind vor ihnen. Beide Methoden betrachten die Nachrichten in der Warteschlange in der Reihenfolge, in der sie eingetroffen sind, und suchen nach einer angegebenen Nachricht (bis zu einem optionalen Timeout). Falls keine solche Meldung `TryScan` wird, gibt `TryScan` None zurück, während `Scan` wartet, bis die Meldung eingeht oder der Vorgang `TryScan` .

Lassen Sie uns es in der Praxis sehen. Wir möchten, dass der Prozessor `RegularOperations` wenn dies möglich ist. Wenn es jedoch eine `PriorityOperation` , sollte diese so schnell wie möglich verarbeitet werden, unabhängig davon, wie viele andere `RegularOperations` in der Warteschlange sind.

```
type Message =
    | RegularOperation of string
    | PriorityOperation of string

let processor = MailboxProcessor<Message>.Start(fun inbox ->
    let rec innerLoop () = async {
        let! priorityData = inbox.TryScan(fun msg ->
            // If there is a PriorityOperation, retrieve its data.
            match msg with
            | PriorityOperation data -> Some data
            | _ -> None)

        match priorityData with
        | Some data ->
            // Process the data from PriorityOperation.
        | None ->
            // No PriorityOperation was in the queue at the time, so
            // let's fall back to processing all possible messages
            let! message = inbox.Receive()
            match message with
            | RegularOperation data ->
```

```
                // We have free time, let's process the RegularOperation.
            | PriorityOperation data ->
                // We did scan the queue, but it might have been empty
                // so it is possible that in the meantime a producer
                // posted a new message and it is a PriorityOperation.
        // And never forget to process next messages.
        innerLoop ()
    }
    innerLoop())
```

Postfachprozessor online lesen: https://riptutorial.com/de/fsharp/topic/9409/postfachprozessor

Kapitel 26: Reflexion

Examples

Robustes Nachdenken durch F # -Zitate

Reflexion ist nützlich, aber fragil. Bedenken Sie:

```
let mi  = typeof<System.String>.GetMethod "StartsWith"
```

Die Probleme mit dieser Art von Code sind:

1. Der Code funktioniert nicht, da `String.StartsWith` mehrere Überladungen `String.StartsWith`
2. Selbst wenn es jetzt keine Überlastungen gibt, können spätere Versionen der Bibliothek eine Überlastung hinzufügen, die einen Laufzeitabsturz verursacht
3. Refactoring-Tools wie `Rename methods` durch Reflektion unterbrochen.

Das heißt, wir bekommen eine Laufzeitabsturz für etwas, das als Kompilierzeit bekannt ist. Das scheint suboptimal zu sein.

Mit `F#` -Zitate können Sie alle oben genannten Probleme vermeiden. Wir definieren einige Hilfsfunktionen:

```
open FSharp.Quotations
open System.Reflection

let getConstructorInfo (e : Expr<'T>) : ConstructorInfo =
  match e with
  | Patterns.NewObject (ci, _) -> ci
  | _ -> failwithf "Expression has the wrong shape, expected NewObject (_, _) instead got: %A"
e


let getMethodInfo (e : Expr<'T>) : MethodInfo =
  match e with
  | Patterns.Call (_, mi, _) -> mi
  | _ -> failwithf "Expression has the wrong shape, expected Call (_, _, _) instead got: %A" e
```

Wir verwenden die Funktionen wie folgt:

```
printfn "%A" <| getMethodInfo <@ "".StartsWith "" @>
printfn "%A" <| getMethodInfo <@ List.singleton 1 @>
printfn "%A" <| getConstructorInfo <@ System.String [||] @>
```

Dies druckt:

```
Boolean StartsWith(System.String)
Void .ctor(Char[])
Microsoft.FSharp.Collections.FSharpList`1[System.Int32] Singleton[Int32](Int32)
```

`<@ ... @>` bedeutet, dass anstelle der Ausführung des Ausdrucks in `F#` eine Ausdrucksstruktur generiert wird, die den Ausdruck darstellt. `<@ "".StartsWith "" @>` generiert eine Ausdrucksbaumstruktur, die folgendermaßen aussieht: `Call (Some (Value ("")), StartsWith, [Value ("")])` . Diese Ausdrucksbaumstruktur entspricht den `getMethodInfo` und gibt die korrekten Methodeninformationen zurück.

Dies löst alle oben aufgeführten Probleme.

Reflexion online lesen: https://riptutorial.com/de/fsharp/topic/4124/reflexion

Kapitel 27: Sequenz

Examples

Sequenzen erzeugen

Es gibt mehrere Möglichkeiten, eine Sequenz zu erstellen.

Sie können Funktionen aus dem Seq-Modul verwenden:

```
// Create an empty generic sequence
let emptySeq = Seq.empty

// Create an empty int sequence
let emptyIntSeq = Seq.empty<int>

// Create a sequence with one element
let singletonSeq = Seq.singleton 10

// Create a sequence of n elements with the specified init function
let initSeq = Seq.init 10 (fun c -> c * 2)

// Combine two sequence to create a new one
let combinedSeq = emptySeq |> Seq.append singletonSeq

// Create an infinite sequence using unfold with generator based on state
let naturals = Seq.unfold (fun state -> Some(state, state + 1)) 0
```

Sie können auch Sequenzausdruck verwenden:

```
// Create a sequence with element from 0 to 10
let intSeq = seq { 0..10 }

// Create a sequence with an increment of 5 from 0 to 50
let intIncrementSeq = seq{ 0..5..50 }

// Create a sequence of strings, yield allow to define each element of the sequence
let stringSeq = seq {
    yield "Hello"
    yield "World"
}

// Create a sequence from multiple sequence, yield! allow to flatten sequences
let flattenSeq = seq {
    yield! seq { 0..10 }
    yield! seq { 11..20 }
}
```

Einführung in Sequenzen

Eine Sequenz ist eine Reihe von Elementen, die aufgezählt werden können. Es ist ein Alias von System.Collections.Generic.IEnumerable und faul. Es speichert eine Reihe von Elementen

desselben Typs (kann ein beliebiger Wert oder ein beliebiges Objekt sein, sogar eine andere Sequenz). Funktionen des Seq.modules können verwendet werden, um darauf zu arbeiten.

Hier ist ein einfaches Beispiel für eine Sequenzaufzählung:

```
let mySeq = { 0..20 } // Create a sequence of int from 0 to 20
mySeq
|> Seq.iter (printf "%i ") // Enumerate each element of the sequence and print it
```

Ausgabe:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Seq.map

```
let seq = seq {0..10}

s |> Seq.map (fun x -> x * 2)

> val it : seq<int> = seq [2; 4; 6; 8; ...]
```

Wenden Sie mit Seq.map eine Funktion auf jedes Element einer Sequenz an

Seq.filter

Angenommen, wir haben eine Folge von ganzen Zahlen und möchten eine Folge erstellen, die nur die geraden ganzen Zahlen enthält. Letzteres können wir über die `filter` des Seq-Moduls erhalten. Die `filter` hat die Typensignatur `('a -> bool) -> seq<'a> -> seq<'a>` ; Dies bedeutet, dass es eine Funktion akzeptiert, die wahr oder falsch (manchmal Prädikat) für eine bestimmte Eingabe des Typs `'a` zurückgibt `'a` und eine Sequenz, die Werte des Typs `'a` , um eine Sequenz zu ergeben, die Werte des Typs `'a` .

```
// Function that tests if an integer is even
let isEven x = (x % 2) = 0

// Generates an infinite sequence that contains the natural numbers
let naturals = Seq.unfold (fun state -> Some(state, state + 1)) 0

// Can be used to filter the naturals sequence to get only the even numbers
let evens = Seq.filter isEven naturals
```

Unendlich wiederholende Sequenzen

```
let data = [1; 2; 3; 4; 5;]
let repeating = seq {while true do yield! data}
```

Wiederholte Sequenzen können mit einem Berechnungsausdruck `seq {}` erstellt werden

Sequenz online lesen: https://riptutorial.com/de/fsharp/topic/2354/sequenz

Kapitel 28: Sequenz-Workflows

Examples

Ertrag und Ertrag!

In Sequenzworkflows fügt `yield` ein einzelnes Element zur erstellten Sequenz hinzu. (In monadischer Terminologie ist es `return` .)

```
> seq { yield 1; yield 2; yield 3 }
val it: seq<int> = seq [1; 2; 3]

> let homogenousTup2ToSeq (a, b) = seq { yield a; yield b }
> tup2Seq ("foo", "bar")
val homogenousTup2ToSeq: 'a * 'a -> seq<'a>
val it: seq<string> = seq ["foo"; "bar"]
```

`yield!` (ausgesprochener *Renditeknall*) fügt alle Elemente einer anderen Sequenz in diese Sequenz ein. Oder mit anderen Worten, es hängt eine Sequenz an. (In Bezug auf Monaden ist es `bind` .)

```
> seq { yield 1; yield! [10;11;12]; yield 20 }
val it: seq<int> = seq [1; 10; 11; 12; 20]

// Creates a sequence containing the items of seq1 and seq2 in order
> let concat seq1 seq2 = seq { yield! seq1; yield! seq2 }
> concat ['a'..'c'] ['x'..'z']
val concat: seq<'a> -> seq<'a> -> seq<'a>
val it: seq<int> = seq ['a'; 'b'; 'c'; 'x'; 'y'; 'z']
```

Sequenzen, die durch Sequenzworkflows erstellt werden, sind auch faul, was bedeutet, dass Elemente der Sequenz nicht wirklich ausgewertet werden, bis sie benötigt werden. Einige Möglichkeiten, Elemente zu erzwingen, umfassen das Aufrufen von `Seq.take` (zieht die ersten n Elemente in eine Sequenz), `Seq.iter` (wendet eine Funktion auf jedes Element zum Ausführen von Nebenwirkungen) oder `Seq.toList` (konvertiert eine Sequenz in eine Liste) . Kombinieren Sie dies mit Rekursion ist, wo `yield!` fängt wirklich an zu glänzen.

```
> let rec numbersFrom n = seq { yield n; yield! numbersFrom (n + 1) }
> let naturals = numbersFrom 0
val numbersFrom: int -> seq<int>
val naturals: seq<int> = seq [0; 1; 2; ...]

// Just like Seq.map: applies a mapping function to each item in a sequence to build a new
sequence
> let rec map f seq1 =
      if Seq.isEmpty seq1 then Seq.empty
      else seq { yield f (Seq.head seq1); yield! map f (Seq.tail seq1) }
> map (fun x -> x * x) [1..10]
val map: ('a -> 'b) -> seq<'a> -> 'b
val it: seq<int> = seq [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

zum

`for` Ausdruck von Sequenzen sieht genauso aus wie sein berühmter Cousin, der Imperativ for-loop. Es "durchläuft" eine Sequenz und wertet den Körper jeder Iteration in der Sequenz aus, die es generiert. Genau wie alles, was mit der Sequenz zusammenhängt, ist es NICHT wandelbar.

```
> let oneToTen = seq { for x in 1..10 -> x }
val oneToTen: seq<int> = seq [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
// Or, equivalently:
> let oneToTen = seq { for x in 1..10 do yield x }
val oneToTen: seq<int> = seq [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

// Just like Seq.map: applies a mapping function to each item in a sequence to build a new
sequence
> let map mapping seq1 = seq { for x in seq1 do yield mapping x }
> map (fun x -> x * x) [1..10]
val map: ('a -> 'b) -> seq<'a> -> seq<'b>
val it: seq<int> = seq [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]

// An infinite sequence of consecutive integers starting at 0
> let naturals =
      let numbersFrom n = seq { yield n; yield! numbersFrom (n + 1) }
      numbersFrom 0
// Just like Seq.filter: returns a sequence consisting only of items from the input sequence
that satisfy the predicate
> let filter predicate seq1 = seq { for x in seq1 do if predicate x then yield x }
> let evenNaturals = naturals |> filter (fun x -> x % 2 = 0)
val naturals: seq<int> = seq [1; 2; 3; ...]
val filter: ('a -> bool) -> seq<'a> -> seq<'a>
val evenNaturals: seq<int> = seq [2; 4; 6; ...]

// Just like Seq.concat: concatenates a collection of sequences together
> let concat seqSeq = seq { for seq in seqSeq do yield! seq }
> concat [[1;2;3];[10;20;30]]
val concat: seq<#seq<'b>> -> seq<'b>
val it: seq<int> = seq [1; 2; 3; 10; 20; 30]
```

Sequenz-Workflows online lesen: https://riptutorial.com/de/fsharp/topic/2785/sequenz-workflows

Kapitel 29: Statisch aufgelöste Typparameter

Syntax

- `s` ist eine Instanz von `^a` Sie zur Kompilierzeit akzeptieren möchten. Dies kann alles sein, was die Mitglieder implementiert, die Sie tatsächlich mit der Syntax aufrufen.
- `^a` ist Generika ähnlich, die `'a` (oder `'A` oder `'T` zum Beispiel) wären, aber diese werden zur Kompilierzeit aufgelöst und erlauben alles, was zu allen angeforderten Verwendungen innerhalb der Methode passt. (keine Schnittstellen erforderlich)

Examples

Einfache Verwendung für alles, was ein Längenmitglied hat

```
let inline getLength s = (^a: (member Length: _) s)
//usage:
getLength "Hello World" // or "Hello World" |> getLength
// returns 11
```

Klasse, Schnittstelle, Datensatznutzung

```
// Record
type Ribbon = {Length:int}
// Class
type Line(len:int) =
    member x.Length = len
type IHaveALength =
    abstract Length:int

let inline getLength s = (^a: (member Length: _) s)
let ribbon = {Length=1}
let line = Line(3)
let someLengthImplementer =
    { new IHaveALength with
        member __.Length = 5}
printfn "Our ribbon length is %i" (getLength ribbon)
printfn "Our Line length is %i" (getLength line)
printfn "Our Object expression length is %i" (getLength someLengthImplementer)
```

Statischer Member-Aufruf

Dies akzeptiert jeden Typ mit einer Methode namens `GetLength` , die nichts übernimmt und ein int zurückgibt:

```
((^a : (static member GetLength : int) ()))
```

Statisch aufgelöste Typparameter online lesen:
https://riptutorial.com/de/fsharp/topic/7228/statisch-aufgeloste-typparameter

Kapitel 30: Typ- und Modulerweiterungen

Bemerkungen

In allen Fällen muss beim Erweitern von Typen und Modulen der Erweiterungscode vor dem Code, der ihn aufrufen soll, hinzugefügt / geladen werden. Es muss auch dem aufrufenden Code zur Verfügung gestellt werden, indem die entsprechenden Namespaces geöffnet / importiert werden .

Examples

Hinzufügen neuer Methoden / Eigenschaften zu vorhandenen Typen

Mit F # können Funktionen zu Typen als "Mitglieder" hinzugefügt werden, wenn sie definiert werden (z. B. Datensatztypen). Mit F # können jedoch auch neue Instanzmitglieder zu bereits *vorhandenen* Typen hinzugefügt werden - auch solche, die an anderer Stelle und in anderen .net-Sprachen deklariert sind.

Im folgenden Beispiel wird allen Instanzen von `String` eine neue Instanzmethode `Duplicate String` .

```
type System.String with
    member this.Duplicate times =
        Array.init times (fun _ -> this)
```

Hinweis : `this` ist ein willkürlich gewählter Variablenname, der verwendet wird, um auf die Instanz des Typs zu verweisen, der erweitert wird - `x` würde genauso gut funktionieren, wäre aber vielleicht weniger selbstbeschreibend.

Es kann dann auf folgende Weise aufgerufen werden.

```
// F#-style call
let result1 = "Hi there!".Duplicate 3

// C#-style call
let result2 = "Hi there!".Duplicate(3)

// Both result in three "Hi there!" strings in an array
```

Diese Funktionalität ist den Erweiterungsmethoden in C # sehr ähnlich.

Neue Eigenschaften können auf dieselbe Weise zu vorhandenen Typen hinzugefügt werden. Sie werden automatisch zu Eigenschaften, wenn das neue Mitglied keine Argumente akzeptiert.

```
type System.String with
    member this.WordCount =
        ' ' // Space character
        |> Array.singleton
        |> fun xs -> this.Split(xs, StringSplitOptions.RemoveEmptyEntries)
```

```
        |> Array.length

let result = "This is an example".WordCount
// result is 4
```

Hinzufügen neuer statischer Funktionen zu vorhandenen Typen

Mit F # können vorhandene Typen mit neuen statischen Funktionen erweitert werden.

```
type System.String with
    static member EqualsCaseInsensitive (a, b) = String.Equals(a, b,
StringComparison.OrdinalIgnoreCase)
```

Diese neue Funktion kann wie folgt aufgerufen werden:

```
let x = String.EqualsCaseInsensitive("abc", "aBc")
// result is True
```

Diese Funktion kann bedeuten, dass Bibliotheken mit Funktionen nicht mehr erstellt werden müssen, sondern sie zu relevanten vorhandenen Typen hinzugefügt werden können. Dies kann nützlich sein, um weitere F # -freundliche Versionen von Funktionen zu erstellen, die Funktionen wie das Currying ermöglichen .

```
type System.String with
    static member AreEqual comparer a b = System.String.Equals(a, b, comparer)

let caseInsensitiveEquals = String.AreEqual StringComparison.OrdinalIgnoreCase

let result = caseInsensitiveEquals "abc" "aBc"
// result is True
```

Hinzufügen von neuen Funktionen zu vorhandenen Modulen und Typen mithilfe von Modulen

Module können verwendet werden, um vorhandenen Modulen und Typen neue Funktionen hinzuzufügen.

```
namespace FSharp.Collections

module List =
    let pair item1 item2 = [ item1; item2 ]
```

Die neue Funktion kann dann aufgerufen werden, als wäre sie ein ursprüngliches Mitglied von List.

```
open FSharp.Collections

module Testing =
    let result = List.pair "a" "b"
    // result is a list containing "a" and "b"
```

Typ- und Modulerweiterungen online lesen: https://riptutorial.com/de/fsharp/topic/2977/typ--und-modulerweiterungen

Kapitel 31: Typen

Examples

Einführung in Typen

Typen können verschiedene Arten von Dingen darstellen. Es können einzelne Daten, ein Datensatz oder eine Funktion sein.

In F # können wir die Typen in zwei Kategorien einteilen:

- F # Typen:

```
// Functions
let a = fun c -> c

// Tuples
let b = (0, "Foo")

// Unit type
let c = ignore

// Records
type r = { Name : string; Age : int }
let d = { Name = "Foo"; Age = 10 }

// Discriminated Unions
type du = | Foo | Bar
let e = Bar

// List and seq
let f = [ 0..10 ]
let g = seq { 0..10 }

// Aliases
type MyAlias = string
```

- .NET-Typen
    - Eingebauter Typ (int, bool, string, ...)
    - Klassen, Strukturen und Schnittstellen
    - Delegierte
    - Arrays

Typ Abkürzungen

Mit Typabkürzungen können Sie Aliase für vorhandene Typen erstellen, um ihnen einen sinnvollen Sinn zu geben.

```
// Name is an alias for a string
type Name = string
```

```
// PhoneNumber is an alias for a string
type PhoneNumber = string
```

Dann können Sie den Alias wie jeden anderen Typ verwenden:

```
// Create a record type with the alias
type Contact = {
    Name : Name
    Phone : PhoneNumber }

// Create a record instance
// We can assign a string since Name and PhoneNumber are just aliases on string type
let c = {
    Name = "Foo"
    Phone = "00 000 000" }

printfn "%A" c

// Output
// {Name = "Foo";
// Phone = "00 000 000";}
```

Seien Sie vorsichtig, Aliase prüfen nicht auf Typkonsistenz. Das bedeutet, dass zwei Aliase, die auf denselben Typ abzielen, einander zugewiesen werden können:

```
let c = {
    Name = "Foo"
    Phone = "00 000 000" }
let d = {
    Name = c.Phone
    Phone = c.Name }

printfn "%A" d

// Output
// {Name = "00 000 000";
// Phone = "Foo";}
```

Typen werden in F # mit dem Schlüsselwort type erstellt

`F#` verwendet das Schlüsselwort `type` , um verschiedene Arten von Typen zu erstellen.

1. Geben Sie Aliase ein
2. Diskriminierte Gewerkschaftstypen
3. Aufnahmetypen
4. Schnittstellentypen
5. Klassenarten
6. Strukturtypen

Beispiele mit gleichwertigem `C#` -Code, soweit möglich:

```
// Equivalent C#:
//  using IntAliasType = System.Int32;
```

```
type IntAliasType = int // As in C# this doesn't create a new type, merely an alias

type DiscriminatedUnionType =
  | FirstCase
  | SecondCase  of int*string

  member x.SomeProperty = // We can add members to DU:s
    match x with
    | FirstCase         -> 0
    | SecondCase (i, _) -> i

type RecordType =
  {
    Id    : int
    Name  : string
  }
  static member New id name : RecordType = // We can add members to records
    { Id = id; Name = name } // { ... } syntax used to create records

// Equivalent C#:
//  interface InterfaceType
//  {
//    int     Id    { get; }
//    string  Name  { get; }
//    int Increment (int i);
//  }
type InterfaceType =
  interface // In order to create an interface type, can also use [<Interface>] attribute
    abstract member Id        : int
    abstract member Name      : string
    abstract member Increment : int -> int
  end

// Equivalent C#:
//  class ClassType : InterfaceType
//  {
//    static int increment (int i)
//    {
//      return i + 1;
//    }
//
//    public ClassType (int id, string name)
//    {
//      Id    = id    ;
//      Name  = name  ;
//    }
//
//    public int    Id    { get; private set; }
//    public string Name  { get; private set; }
//    public int   Increment (int i)
//    {
//      return increment (i);
//    }
//  }
type ClassType (id : int, name : string) = // a class type requires a primary constructor
  let increment i = i + 1 // Private helper functions

  interface InterfaceType with // Implements InterfaceType
    member x.Id           = id
    member x.Name         = name
    member x.Increment i  = increment i
```

```
// Equivalent C#:
//  class SubClassType : ClassType
//  {
//    public SubClassType (int id, string name) : base(id, name)
//    {
//    }
//  }
type SubClassType (id : int, name : string) =
  inherit ClassType (id, name) // Inherits ClassType

// Equivalent C#:
//  struct StructType
//  {
//    public StructType (int id)
//    {
//      Id = id;
//    }
//
//    public int Id { get; private set; }
//  }
type StructType (id : int) =
  struct  // In order create a struct type, can also use [<Struct>] attribute
    member x.Id = id
  end
```

Typ Inferenz

Wissen

Dieses Beispiel wurde aus diesem Artikel zur Typeninferenz angepasst

Was ist Typ Inferenz?

Type Inference ist der Mechanismus, mit dem der Compiler ableiten kann, welche Typen wo verwendet werden. Dieser Mechanismus basiert auf einem Algorithmus, der häufig als "Hindley-Milner" oder "HM" bezeichnet wird. Nachfolgend einige Regeln zum Ermitteln der Typen von einfachen Werten und Funktionswerten:

- Schau dir die Literale an
- Schauen Sie sich die Funktionen und andere Werte an, mit denen etwas interagiert
- Sehen Sie sich explizite Typeinschränkungen an
- Wenn es nirgends irgendwelche Einschränkungen gibt, generalisieren Sie automatisch auf generische Typen

Schau dir die Literale an

Der Compiler kann Typen anhand der Literale ableiten. Wenn das Literal ein int ist und Sie "x" hinzufügen, muss "x" ebenfalls ein int sein. Wenn das Literal jedoch ein Float ist und Sie "x" hinzufügen, muss "x" ebenfalls ein Float sein.

Hier sind einige Beispiele:

```
let inferInt x = x + 1
let inferFloat x = x + 1.0
let inferDecimal x = x + 1m     // m suffix means decimal
let inferSByte x = x + 1y       // y suffix means signed byte
let inferChar x = x + 'a'       // a char
let inferString x = x + "my string"
```

Schauen Sie sich die Funktionen und andere Werte an, mit denen sie interagieren

Wenn nirgendwo Literale vorhanden sind, versucht der Compiler, die Typen zu ermitteln, indem er die Funktionen und andere Werte analysiert, mit denen sie interagieren.

```
let inferInt x = x + 1
let inferIndirectInt x = inferInt x       //deduce that x is an int

let inferFloat x = x + 1.0
let inferIndirectFloat x = inferFloat x   //deduce that x is a float

let x = 1
let y = x     //deduce that y is also an int
```

Sehen Sie sich explizite Typeinschränkungen oder Anmerkungen an

Wenn explizite Typeinschränkungen oder Anmerkungen angegeben wurden, verwendet der Compiler diese.

```
let inferInt2 (x:int) = x                // Take int as parameter
let inferIndirectInt2 x = inferInt2 x    // Deduce from previous that x is int

let inferFloat2 (x:float) = x                // Take float as parameter
let inferIndirectFloat2 x = inferFloat2 x    // Deduce from previous that x is float
```

Automatische Generalisierung

Wenn nach all dem keine Einschränkungen gefunden werden, macht der Compiler die Typen nur zu generisch.

```
let inferGeneric x = x
let inferIndirectGeneric x = inferGeneric x
let inferIndirectGenericAgain x = (inferIndirectGeneric x).ToString()
```

Dinge, die durch Typeninferenz schief gehen können

Der Typ Inferenz ist leider nicht perfekt. Manchmal hat der Compiler keine Ahnung, was er tun soll. Wenn Sie wissen, was passiert, können Sie wirklich ruhig bleiben, anstatt den Compiler töten zu wollen. Hier sind einige der Hauptgründe für Typfehler:

- Erklärungen außer Betrieb
- Nicht genug Information
- Überlastete Methoden

Erklärungen außer Betrieb

Eine grundlegende Regel ist, dass Sie Funktionen deklarieren müssen, bevor sie verwendet werden.

Dieser Code schlägt fehl:

```
let square2 x = square x   // fails: square not defined
let square x = x * x
```

Das ist aber ok:

```
let square x = x * x
let square2 x = square x   // square already defined earlier
```

Rekursive oder gleichzeitige Deklarationen

Eine Variante des Problems "Außer Betrieb" tritt bei rekursiven Funktionen oder Definitionen auf, die sich aufeinander beziehen müssen. In diesem Fall hilft keine Neuordnung. Wir müssen zusätzliche Schlüsselwörter verwenden, um den Compiler zu unterstützen.

Wenn eine Funktion kompiliert wird, ist der Funktionsbezeichner für den Körper nicht verfügbar. Wenn Sie also eine einfache rekursive Funktion definieren, wird ein Compiler-Fehler angezeigt. Das Update besteht darin, das Schlüsselwort "rec" als Teil der Funktionsdefinition hinzuzufügen. Zum Beispiel:

```
// the compiler does not know what "fib" means
let fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
   // error FS0039: The value or constructor 'fib' is not defined
```

Hier ist die feste Version mit "rec fib" hinzugefügt, um anzuzeigen, dass sie rekursiv ist:

```
let rec fib n =              // LET REC rather than LET
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

Nicht genug Information

Manchmal hat der Compiler nicht genügend Informationen, um einen Typ zu bestimmen. Im folgenden Beispiel weiß der Compiler nicht, mit welchem Typ die Length-Methode arbeiten soll. Aber es kann es auch nicht generisch machen, klagt es.

```
let stringLength s = s.Length
  // error FS0072: Lookup on object of indeterminate type
  // based on information prior to this program point.
  // A type annotation may be needed ...
```

Diese Fehler können durch explizite Anmerkungen behoben werden.

```
let stringLength (s:string) = s.Length
```

Überlastete Methoden

Wenn Sie eine externe Klasse oder Methode in .NET aufrufen, erhalten Sie häufig Fehler aufgrund von Überladung.

In vielen Fällen, z. B. im folgenden concat-Beispiel, müssen Sie die Parameter der externen Funktion explizit kommentieren, damit der Compiler weiß, welche überladene Methode aufgerufen wird.

```
let concat x = System.String.Concat(x)           //fails
let concat (x:string) = System.String.Concat(x)  //works
let concat x = System.String.Concat(x:string)    //works
```

Manchmal haben die überladenen Methoden andere Argumentnamen. In diesem Fall können Sie dem Compiler auch einen Hinweis geben, indem Sie die Argumente benennen. Hier ist ein Beispiel für den StreamReader-Konstruktor.

```
let makeStreamReader x = new System.IO.StreamReader(x)        //fails
let makeStreamReader x = new System.IO.StreamReader(path=x)   //works
```

Typen online lesen: https://riptutorial.com/de/fsharp/topic/3559/typen

Kapitel 32: Verwenden von F #, WPF, FsXaml, einem Menü und einem Dialogfeld

Einführung

Ziel ist es, eine einfache Anwendung in F # mit Windows Presentation Foundation (WPF) mit herkömmlichen Menüs und Dialogfeldern zu erstellen. Es rührt von meiner Frustration her, als ich versuchte, hunderte Abschnitte der Dokumentation, Artikel und Beiträge zu F # und WPF durchzugehen. Um irgendetwas mit WPF machen zu können, müssen Sie anscheinend alles darüber wissen. Ich möchte hier einen möglichen Weg aufzeigen, ein einfaches Desktop-Projekt, das als Vorlage für Ihre Apps dienen kann.

Examples

Richten Sie das Projekt ein

Wir gehen davon aus, dass Sie dies in Visual Studio 2015 tun (in meinem Fall in der VS 2015-Community). Erstellen Sie ein leeres Konsolenprojekt in VS. In Projekt | Eigenschaften ändern den Ausgabetyp in Windows-Anwendung.

Verwenden Sie als Nächstes NuGet, um FsXaml.Wpf zum Projekt hinzuzufügen. Dieses Paket wurde vom geschätzten Reed Copsey Jr. erstellt und vereinfacht die Verwendung von WPF von F # erheblich. Bei der Installation werden eine Reihe weiterer WPF-Assemblys hinzugefügt, sodass Sie dies nicht tun müssen. Es gibt andere ähnliche Pakete zu FsXaml, aber eines meiner Ziele bestand darin, die Anzahl der Tools so gering wie möglich zu halten, um das Gesamtprojekt so einfach und wartbar wie möglich zu gestalten.

Fügen Sie außerdem UIAutomationTypes als Referenz hinzu. es kommt als Teil von .NET.

Fügen Sie die "Geschäftslogik" hinzu

Vermutlich wird Ihr Programm etwas tun. Fügen Sie Ihrem Projekt Ihren Arbeitscode anstelle von Program.fs hinzu. In diesem Fall ist es unsere Aufgabe, Spirographenkurven auf einem Fensterbereich zu zeichnen. Dies wird mit Spirograph.fs unten erreicht.

```
namespace Spirograph

// open System.Windows does not automatically open all its sub-modules, so we
// have to open them explicitly as below, to get the resources noted for each.
open System                             // for Math.PI
open System.Windows                     // for Point
open System.Windows.Controls            // for Canvas
open System.Windows.Shapes              // for Ellipse
open System.Windows.Media               // for Brushes

// ------------------------------------------------------------------------
```

```
// This file is first in the build sequence, so types should be defined here
type DialogBoxXaml  = FsXaml.XAML<"DialogBox.xaml">
type MainWindowXaml = FsXaml.XAML<"MainWindow.xaml">
type App            = FsXaml.XAML<"App.xaml">

// ----------------------------------------------------------------------------
// Model: This draws the Spirograph
type MColor = | MBlue   | MRed | MRandom

type Model() =
  let mutable myCanvas: Canvas = null
  let mutable myR              = 220    // outer circle radius
  let mutable myr              = 65     // inner circle radius
  let mutable myl              = 0.8    // pen position relative to inner circle
  let mutable myColor          = MBlue  // pen color

  let rng                      = new Random()
  let mutable myRandomColor    = Color.FromRgb(rng.Next(0, 255) |> byte,
                                               rng.Next(0, 255) |> byte,
                                               rng.Next(0, 255) |> byte)

  member this.MyCanvas
    with get() = myCanvas
    and  set(newCanvas) = myCanvas <- newCanvas

  member this.MyR
    with get() = myR
    and  set(newR) = myR <- newR

  member this.Myr
    with get() = myr
    and  set(newr) = myr <- newr

  member this.Myl
    with get() = myl
    and  set(newl) = myl <- newl

  member this.MyColor
    with get() = myColor
    and  set(newColor) = myColor <- newColor

  member this.Randomize =
    // Here we randomize the parameters. You can play with the possible ranges of
    // the parameters to find randomized spirographs that are pleasing to you.
    this.MyR      <- rng.Next(100, 500)
    this.Myr      <- rng.Next(this.MyR / 10, (9 * this.MyR) / 10)
    this.Myl      <- 0.1 + 0.8 * rng.NextDouble()
    this.MyColor  <- MRandom
    myRandomColor <- Color.FromRgb(rng.Next(0, 255) |> byte,
                                   rng.Next(0, 255) |> byte,
                                   rng.Next(0, 255) |> byte)

  member this.DrawSpirograph =
    // Draw a spirograph. Note there is some fussing with ints and floats; this
    // is required because the outer and inner circle radii are integers. This is
    // necessary in order for the spirograph to return to its starting point
    // after a certain number of revolutions of the outer circle.

    // Start with usual recursive gcd function and determine the gcd of the inner
    // and outer circle radii. Everything here should be in integers.
    let rec gcd x y =
```

```
        if y = 0 then x
        else gcd y (x % y)

    let g = gcd this.MyR this.Myr             // find greatest common divisor
    let maxRev = this.Myr / g                 // maximum revs to repeat

    // Determine width and height of window, location of center point, scaling
    // factor so that spirograph fits within the window, ratio of inner and outer
    // radii.

    // Everything from this point down should be float.
    let width, height = myCanvas.ActualWidth, myCanvas.ActualHeight
    let cx, cy = width / 2.0, height / 2.0    // coordinates of center point
    let maxR   = min cx cy                    // maximum radius of outer circle
    let scale  = maxR / float(this.MyR)             // scaling factor
    let rRatio = float(this.Myr) / float(this.MyR)  // ratio of the radii

    // Build the collection of spirograph points, scaled to the window.
    let points = new PointCollection()
    for degrees in [0 .. 5 .. 360 * maxRev] do
      let angle = float(degrees) * Math.PI / 180.0
      let x, y = cx + scale * float(this.MyR) *
                 ((1.0-rRatio)*Math.Cos(angle) +
                  this.Myl*rRatio*Math.Cos((1.0-rRatio)*angle/rRatio)),
                 cy + scale * float(this.MyR) *
                 ((1.0-rRatio)*Math.Sin(angle) -
                  this.Myl*rRatio*Math.Sin((1.0-rRatio)*angle/rRatio))
      points.Add(new Point(x, y))

    // Create the Polyline with the above PointCollection, erase the Canvas, and
    // add the Polyline to the Canvas Children
    let brush = match this.MyColor with
                | MBlue   -> Brushes.Blue
                | MRed    -> Brushes.Red
                | MRandom -> new SolidColorBrush(myRandomColor)

    let mySpirograph = new Polyline()
    mySpirograph.Points <- points
    mySpirograph.Stroke <- brush

    myCanvas.Children.Clear()
    this.MyCanvas.Children.Add(mySpirograph) |> ignore
```

Spirograph.fs ist die erste F # -Datei in der Kompilierreihenfolge. Sie enthält also die Definitionen der benötigten Typen. Seine Aufgabe ist es, ein Spirograph auf der Leinwand auf der Grundlage der in einem Dialogfeld eingegebenen Parameter zu zeichnen. Da es viele Hinweise zum Zeichnen eines Spirographen gibt, gehen wir hier nicht darauf ein.

Erstellen Sie das Hauptfenster in XAML

Sie müssen eine XAML-Datei erstellen, die das Hauptfenster definiert, das unser Menü und den Zeichenbereich enthält. Hier ist der XAML-Code in MainWindow.xaml:

```
<!-- This defines the main window, with a menu and a canvas. Note that the Height
     and Width are overridden in code to be 2/3 the dimensions of the screen -->
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Spirograph" Height="200" Width="300">
    <!-- Define a grid with 3 rows: Title bar, menu bar, and canvas. By default
         there is only one column -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <!-- Define the menu entries -->
        <Menu  Grid.Row="0">
            <MenuItem Header="File">
                <MenuItem Header="Exit"
                          Name="menuExit"/>
            </MenuItem>
            <MenuItem Header="Spirograph">
                <MenuItem Header="Parameters..."
                          Name="menuParameters"/>
                <MenuItem Header="Draw"
                          Name="menuDraw"/>
            </MenuItem>
            <MenuItem Header="Help">
                <MenuItem Header="About"
                          Name="menuAbout"/>
            </MenuItem>
        </Menu>
        <!-- This is a canvas for drawing on. If you don't specify the coordinates
             for Left and Top you will get NaN for those values -->
        <Canvas Grid.Row="1" Name="myCanvas" Left="0" Top="0">
        </Canvas>
    </Grid>
</Window>
```

Kommentare sind normalerweise nicht in XAML-Dateien enthalten, was ich für einen Fehler halte. Ich habe allen XAML-Dateien in diesem Projekt einige Kommentare hinzugefügt. Ich behaupte nicht, dass dies die besten Kommentare sind, die jemals geschrieben wurden, aber sie zeigen zumindest, wie ein Kommentar formatiert werden sollte. Beachten Sie, dass geschachtelte Kommentare in XAML nicht zulässig sind.

Erstellen Sie das Dialogfeld in XAML und F

Die XAML-Datei für die Spirograph-Parameter befindet sich unten. Es enthält drei Textfelder für die Spirograph-Parameter und eine Gruppe von drei Optionsfeldern für Farbe. Wenn wir Optionsfeldern den gleichen Gruppennamen zuweisen, übernimmt WPF das Ein- und Ausschalten, wenn ausgewählt wird.

```
<!-- This first part is boilerplate, except for the title, height and width.
     Note that some fussing with alignment and margins may be required to get
     the box looking the way you want it. -->
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Parameters" Height="200" Width="250">
    <!-- Here we define a layout of 3 rows and 2 columns below the title bar -->
    <Grid>
        <Grid.RowDefinitions>
```

```
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <!-- Define a label and a text box for the first three rows. Top row is
         the integer radius of the outer circle -->
    <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="0"
                Grid.ColumnSpan="2">
        <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="R: Outer"
               Height="24" Width='65'/>
        <TextBox x:Name="radiusR"  Margin="0,0,0,0.5" Width="120"
                 VerticalAlignment="Bottom" Height="20">Integer</TextBox>
    </StackPanel>
    <!-- This defines a label and text box for the integer radius of the
         inner circle -->
    <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="1"
                Grid.ColumnSpan="2">
        <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="r: Inner"
               Height="24" Width='65'/>
        <TextBox x:Name="radiusr"  Margin="0,0,0,0.5" Width="120"
                 VerticalAlignment="Bottom" Height="20" Text="Integer"/>
    </StackPanel>
    <!-- This defines a label and text box for the float ratio of the inner
         circle radius at which the pen is positioned -->
    <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="2"
                Grid.ColumnSpan="2">
        <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="l: Ratio"
               Height="24" Width='65'/>
        <TextBox x:Name="ratiol"  Margin="0,0,0,1" Width="120"
                 VerticalAlignment="Bottom" Height="20" Text="Float"/>
    </StackPanel>
    <!-- This defines a radio button group to select color -->
    <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="3"
                Grid.ColumnSpan="2">
        <Label VerticalAlignment="Top" Margin="5,6,4,5.333" Content="Color"
               Height="24"/>
        <RadioButton x:Name="buttonBlue" Content="Blue" GroupName="Color"
                     HorizontalAlignment="Left"  VerticalAlignment="Top"
                     Click="buttonBlueClick"
                     Margin="5,13,11,3.5" Height="17"/>
        <RadioButton x:Name="buttonRed"  Content="Red"  GroupName="Color"
                     HorizontalAlignment="Left" VerticalAlignment="Top"
                     Click="buttonRedClick"
                     Margin="5,13,5,3.5" Height="17" />
        <RadioButton x:Name="buttonRandom"  Content="Random"
                     GroupName="Color" Click="buttonRandomClick"
                     HorizontalAlignment="Left" VerticalAlignment="Top"
                     Margin="5,13,5,3.5" Height="17" />
    </StackPanel>
    <!-- These are the standard OK/Cancel buttons -->
    <Button Grid.Row="4" Grid.Column="0" Name="okButton"
            Click="okButton_Click" IsDefault="True">OK</Button>
    <Button Grid.Row="4" Grid.Column="1" Name="cancelButton"
            IsCancel="True">Cancel</Button>
</Grid>
```

```
</Window>
```

Jetzt fügen wir den Code für die Dialog.Box hinzu. Der Code, der für die Behandlung der Schnittstelle des Dialogfelds mit dem Rest des Programms verwendet wird, heißt standardmäßig XXX.xaml.fs, wobei die zugehörige XAML-Datei XXX.xaml heißt.

```
namespace Spirograph

open System.Windows.Controls

type DialogBox(app: App, model: Model, win: MainWindowXaml) as this =
  inherit DialogBoxXaml()

  let myApp   = app
  let myModel = model
  let myWin   = win

  // These are the default parameters for the spirograph, changed by this dialog
  // box
  let mutable myR = 220                // outer circle radius
  let mutable myr = 65                 // inner circle radius
  let mutable myl = 0.8                // pen position relative to inner circle
  let mutable myColor = MBlue          // pen color

  // These are the dialog box controls. They are initialized when the dialog box
  // is loaded in the whenLoaded function below.
  let mutable RBox: TextBox = null
  let mutable rBox: TextBox = null
  let mutable lBox: TextBox = null

  let mutable blueButton: RadioButton   = null
  let mutable redButton: RadioButton    = null
  let mutable randomButton: RadioButton = null

  // Call this functions to enable or disable parameter input depending on the
  // state of the randomButton. This is a () -> () function to keep it from
  // being executed before we have loaded the dialog box below and found the
  // values of TextBoxes and RadioButtons.
  let enableParameterFields(b: bool) =
    RBox.IsEnabled <- b
    rBox.IsEnabled <- b
    lBox.IsEnabled <- b

  let whenLoaded _ =
    // Load and initialize text boxes and radio buttons to the current values in
    // the model. These are changed only if the OK button is clicked, which is
    // handled below. Also, if the color is Random, we disable the parameter
    // fields.
    RBox <- this.FindName("radiusR") :?> TextBox
    rBox <- this.FindName("radiusr") :?> TextBox
    lBox <- this.FindName("ratiol")  :?> TextBox

    blueButton   <- this.FindName("buttonBlue")   :?> RadioButton
    redButton    <- this.FindName("buttonRed")    :?> RadioButton
    randomButton <- this.FindName("buttonRandom") :?> RadioButton

    RBox.Text <- myModel.MyR.ToString()
    rBox.Text <- myModel.Myr.ToString()
    lBox.Text <- myModel.Myl.ToString()
```

```
  myR <- myModel.MyR
  myr <- myModel.Myr
  myl <- myModel.Myl

  blueButton.IsChecked   <- new System.Nullable<bool>(myModel.MyColor = MBlue)
  redButton.IsChecked    <- new System.Nullable<bool>(myModel.MyColor = MRed)
  randomButton.IsChecked <- new System.Nullable<bool>(myModel.MyColor = MRandom)

  myColor <- myModel.MyColor
  enableParameterFields(not (myColor = MRandom))

let whenClosing _ =
  // Show the actual spirograph parameters in a message box at close. Note the
  // \n in the sprintf gives us a linebreak in the MessageBox. This is mainly
  // for debugging, and it can be deleted.
  let s = sprintf "R = %A\nr = %A\nl = %A\nColor = %A"
                  myModel.MyR myModel.Myr myModel.Myl myModel.MyColor
  System.Windows.MessageBox.Show(s, "Spirograph") |> ignore
  ()

let whenClosed _ =
  ()

do
  this.Loaded.Add whenLoaded
  this.Closing.Add whenClosing
  this.Closed.Add whenClosed

override this.buttonBlueClick(sender: obj,
                              eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MBlue
  enableParameterFields(true)
  ()

override this.buttonRedClick(sender: obj,
                             eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MRed
  enableParameterFields(true)
  ()

override this.buttonRandomClick(sender: obj,
                                eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MRandom
  enableParameterFields(false)
  ()

override this.okButton_Click(sender: obj,
                             eArgs: System.Windows.RoutedEventArgs) =
  // Only change the spirograph parameters in the model if we hit OK in the
  // dialog box.
  if myColor = MRandom
  then myModel.Randomize
  else myR <- RBox.Text |> int
       myr <- rBox.Text |> int
       myl <- lBox.Text |> float

       myModel.MyR   <- myR
       myModel.Myr   <- myr
       myModel.Myl   <- myl
       model.MyColor <- myColor
```

```
    // Note that setting the DialogResult to nullable true is essential to get
    // the OK button to work.
    this.DialogResult <- new System.Nullable<bool> true
    ()
```

Ein Großteil des Codes dient dazu, sicherzustellen, dass die Spirograph-Parameter in Spirograph.fs mit denen in diesem Dialogfeld übereinstimmen. Beachten Sie, dass keine Fehlerprüfung erfolgt: Wenn Sie einen Gleitkommawert für die in den beiden oberen Parameterfeldern erwarteten Ganzzahlen eingeben, stürzt das Programm ab. Bitte fügen Sie Fehlerprüfung in Ihren eigenen Aufwand hinzu.

Beachten Sie auch, dass die Parametereingabefelder deaktiviert sind. In den Optionsschaltflächen wird die Farbe Zufällig ausgewählt. Es ist hier nur um zu zeigen, wie es geht.

Um Daten zwischen dem Dialogfeld und dem Programm hin und her zu verschieben, benutze ich System.Windows.Element.FindName (), um das entsprechende Steuerelement zu suchen, um es in das Steuerelement umzuwandeln, das es sein soll, und dann die entsprechenden Einstellungen von der Steuerung. Die meisten anderen Beispielprogramme verwenden Datenbindung. Ich habe es aus zwei Gründen nicht getan: Erstens konnte ich nicht herausfinden, wie ich es schaffen kann, und zweitens bekam ich, wenn es nicht funktionierte, keine Fehlermeldung. Vielleicht kann mir jemand, der dies auf StackOverflow besucht, erklären, wie die Datenbindung verwendet wird, ohne eine ganze Reihe neuer NuGet-Pakete hinzuzufügen.

Fügen Sie den Code hinter MainWindow.xaml hinzu

```
namespace Spirograph

type MainWindow(app: App, model: Model) as this =
  inherit MainWindowXaml()

  let myApp   = app
  let myModel = model

  let whenLoaded _ =
    ()

  let whenClosing _ =
    ()

  let whenClosed _ =
    ()

  let menuExitHandler _ =
    System.Windows.MessageBox.Show("Good-bye", "Spirograph") |> ignore
    myApp.Shutdown()
    ()

  let menuParametersHandler _ =
    let myParametersDialog = new DialogBox(myApp, myModel, this)
    myParametersDialog.Topmost <- true
    let bResult = myParametersDialog.ShowDialog()
    myModel.DrawSpirograph
    ()
```

```
  let menuDrawHandler _ =
    if myModel.MyColor = MRandom then myModel.Randomize
    myModel.DrawSpirograph
    ()

  let menuAboutHandler _ =
    System.Windows.MessageBox.Show("F#/WPF Menus & Dialogs", "Spirograph")
    |> ignore
    ()

  do
    this.Loaded.Add whenLoaded
    this.Closing.Add whenClosing
    this.Closed.Add whenClosed
    this.menuExit.Click.Add menuExitHandler
    this.menuParameters.Click.Add menuParametersHandler
    this.menuDraw.Click.Add menuDrawHandler
    this.menuAbout.Click.Add menuAboutHandler
```

Hier ist nicht viel los: Wir öffnen bei Bedarf das Dialogfeld "Parameter" und können den Spirographen mit den aktuellen Parametern neu zeichnen.

Fügen Sie App.xaml und App.xaml.fs hinzu, um alles miteinander zu verknüpfen

```
<!-- All boilerplate for now -->
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Hier ist der Code dahinter:

```
namespace Spirograph

open System
open System.Windows
open System.Windows.Controls

module Main =
  [<STAThread; EntryPoint>]
  let main _ =
    // Create the app and the model with the "business logic", then create the
    // main window and link its Canvas to the model so the model can access it.
    // The main window is linked to the app in the Run() command in the last line.
    let app = App()
    let model = new Model()
    let mainWindow = new MainWindow(app, model)
    model.MyCanvas <- (mainWindow.FindName("myCanvas") :?> Canvas)

    // Make sure the window is on top, and set its size to 2/3 of the dimensions
    // of the screen.
    mainWindow.Topmost <- true
    mainWindow.Height  <-
```

```
    (System.Windows.SystemParameters.PrimaryScreenHeight * 0.67)
mainWindow.Width   <-
    (System.Windows.SystemParameters.PrimaryScreenWidth * 0.67)

app.Run(mainWindow) // Returns application's exit code.
```

App.xaml ist hier nur eine Zwischenablage, hauptsächlich um zu zeigen, wo Anwendungsressourcen wie Icons, Grafiken oder externe Dateien deklariert werden können. Der Begleiter App.xaml.fs fasst Model und MainWindow zusammen, formatiert MainWindow auf zwei Drittel der verfügbaren Bildschirmgröße und führt es aus.

Wenn Sie dies erstellen, müssen Sie sicherstellen, dass die Build-Eigenschaft für jede XAML-Datei auf Resource gesetzt ist. Dann können Sie entweder den Debugger ausführen oder eine Exe-Datei erstellen. Beachten Sie, dass Sie dies nicht mit dem F # -Interpreter ausführen können: Das FsXaml-Paket und der Interpreter sind nicht kompatibel.

Hier hast du es. Ich hoffe, Sie können dies als Ausgangspunkt für Ihre eigenen Anwendungen nutzen und können dadurch Ihr Wissen über das hier gezeigte hinaus erweitern. Alle Anmerkungen und Vorschläge werden geschätzt.

Verwenden von F #, WPF, FsXaml, einem Menü und einem Dialogfeld online lesen: https://riptutorial.com/de/fsharp/topic/9145/verwenden-von-f-sharp--wpf--fsxaml--einem-menu-und-einem-dialogfeld

Kapitel 33: Zeichenketten

Examples

String-Literale

```
let string1 = "Hello" //simple string

let string2 = "Line\nNewLine" //string with newline escape sequence

let string3 = @"Line\nSameLine" //use @ to create a verbatim string literal

let string4 = @"Line""with""quoutes inside" //double quoute to indicate a single quoute inside
@ string

let string5 = """single "quote" is ok""" //triple-quote string literal, all symbol including
quote are verbatim

let string6 = "ab
cd"// same as "ab\ncd"

let string7 = "xx\
    yy" //same as "xxyy", backslash at the end contunies the string without new line, leading
whitespace on the next line is ignored
```

Einfache String-Formatierung

Es gibt mehrere Möglichkeiten, einen String zu formatieren und als Ergebnis zu erhalten.

Die .NET-Methode ist mit `String.Format` oder `StringBuilder.AppendFormat` :

```
open System
open System.Text

let hello = String.Format ("Hello {0}", "World")
// return a string with "Hello World"

let builder = StringBuilder()
let helloAgain = builder.AppendFormat ("Hello {0} again!", "World")
// return a StringBuilder with "Hello World again!"
```

F # hat auch Funktionen zum Formatieren von Zeichenfolgen im C-Stil. Für jede .NET-Funktion gibt es Äquivalente:

- `sprintf` (String.Format):

```
open System

let hello = sprintf "Hello %s" "World"
// "Hello World", "%s" is for string

let helloInt = sprintf "Hello %i" 42
```

```
// "Hello 42", "%i" is for int

let helloFloat = sprintf "Hello %f" 4.2
// "Hello 4.2000", "%f" is for float

let helloBool = sprintf "Hello %b" true
// "Hello true", "%b" is for bool

let helloNativeType = sprintf "Hello %A again!" ("World", DateTime.Now)
// "Hello {formatted date}", "%A" is for native type

let helloObject = sprintf "Hello %O again!" DateTime.Now
// "Hello {formatted date}", "%O" is for calling ToString
```

- `bprintf` (StringBuilder.AppendFormat):

```
open System
open System.Text

let builder = StringBuilder()

// Attach the StringBuilder to the format function with partial application
let append format = Printf.bprintf builder format

// Same behavior as sprintf but strings are appended to a StringBuilder
append "Hello %s again!\n" "World"
append "Hello %i again!\n" 42
append "Hello %f again!\n" 4.2
append "Hello %b again!\n" true
append "Hello %A again!\n" ("World", DateTime.Now)
append "Hello %O again!\n" DateTime.Now

builder.ToString() // Get the result string
```

Die Verwendung dieser Funktionen anstelle der .NET-Funktionen bietet einige Vorteile:

- Geben Sie Sicherheit ein
- Teilanwendung
- F # native Typunterstützung

Zeichenketten online lesen: https://riptutorial.com/de/fsharp/topic/1397/zeichenketten

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit F #	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	1: F # WPF-Code hinter der Anwendung mit FsXaml	Bent Tranberg
3	Aktive Muster	Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
4	Aufzeichnungen	eirik, goric, Ringil
5	Der Typ "Einheit"	4444, Abel, Jake Lishman, rmunn
6	Designmuster-Implementierung in F #	FuleSnabel, Ringil
7	Diskriminierte Gewerkschaften	chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen, Vandroiy, VillasV
8	Einführung in WPF in F #	Funk
9	F # auf .NET Core	Boggin, Joel Martinez
10	F # Tipps und Tricks zur Leistung	FuleSnabel, Paul Westcott, Ringil, s952163
11	Falten	Jean-Claude Colette, Zaid Ajaj
12	Faule Bewertung	inzi
13	Funktionen	asibahi, Julien Pires, rmunn, ronilk
14	Geben Sie Anbieter ein	GregC, jdphenix, Joel Martinez
15	Generics	Jake Lishman
16	Klassen	asibahi, inzi, RamenChef, Tomasz Maczyński

17	Listen	asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj
18	Maßeinheiten	asibahi, goric, GregC, Vandroiy
19	Memoization	Jean-Claude Colette, Julien Pires, Ringil
20	Monaden	FuleSnabel
21	Musterabgleich	asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene
22	Operatoren	FuleSnabel
23	Optionstypen	asibahi, chillitom, FuleSnabel
24	Portierung von C # nach F #	jdphenix, marklam, RamenChef
25	Postfachprozessor	Honza Brestan
26	Reflexion	FuleSnabel
27	Sequenz	Foggy Finder, inzi, James McCalden, Julien Pires, s952163
28	Sequenz-Workflows	Jwosty
29	Statisch aufgelöste Typparameter	Maslow
30	Typ- und Modulerweiterungen	Jono Job
31	Typen	Cedric Royer-Bertrand, FuleSnabel, Julien Pires
32	Verwenden von F #, WPF, FsXaml, einem Menü und einem Dialogfeld	Bob McCrory, Goswin
33	Zeichenketten	FireAlkazar, Julien Pires