EBook Gratuito

APPRENDIMENTO F#

Free unaffiliated eBook created from
Stack Overflow contributors.

#f#

Sommario

Di

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

Capitolo 1: Iniziare con F

Osservazioni

F # è un linguaggio "funzionale-primo". Puoi conoscere tutti i diversi tipi di espressioni , insieme alle funzioni .

Il compilatore F #, che è open source , compila i tuoi programmi in IL, il che significa che puoi usare il codice F # da qualsiasi linguaggio compatibile con .NET come C # ; ed eseguilo su Mono, .NET Core o .NET Framework su Windows.

Versioni

Versione	Data di rilascio
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

Installazione o configurazione

finestre

Se hai installato Visual Studio (qualsiasi versione compresa express e community), F # dovrebbe essere già incluso. Basta scegliere F # come lingua quando si crea un nuovo progetto. Oppure vedi http://fsharp.org/use/windows/ per ulteriori opzioni.

OS X

Xamarin Studio supporta F #. In alternativa, è possibile utilizzare VS Code per OS X , che è un editor multipiattaforma di Microsoft.
Una volta terminato con l'installazione di VS Code, avviare `VS Code Quick Open` (`Ctrl` + `P`) quindi eseguire `ext install Ionide-fsharp`

Potresti anche considerare Visual Studio per Mac .

Ci sono altre alternative qui descritte .

Linux

Installa i pacchetti `mono-complete` e `fsharp` tramite il gestore pacchetti della tua distribuzione (Apt, Yum, ecc.). Per una buona esperienza di editing, utilizzare Visual Studio Code e installare il `ionide-fsharp` , oppure utilizzare Atom e installare il `ionide-installer` . Vedi http://fsharp.org/use/linux/ per ulteriori opzioni.

Ciao mondo!

Questo è il codice per un semplice progetto di console, che stampa "Hello, World!" a STDOUT ed esce con un codice di uscita di `0`

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

Esempio di disaggregazione linea per linea:

- `[<EntryPoint>]` - Un attributo .net che contrassegna "il metodo che usi per impostare il punto di ingresso" del tuo programma (fonte).
- `let main argv` - definisce una funzione chiamata `main` con un singolo parametro `argv` . Poiché questo è il punto di ingresso del programma, `argv` sarà una serie di stringhe. I contenuti dell'array sono gli argomenti che sono stati passati al programma quando è stato eseguito.
- `printfn "Hello, World!"` - la funzione `printfn` restituisce la stringa ** passata come primo argomento, aggiungendo anche una nuova riga.
- `0` funzioni `0` - F # restituiscono sempre un valore e il valore restituito è il risultato dell'ultima espressione nella funzione. Mettere `0` come ultima riga significa che la funzione restituirà sempre zero (un numero intero).

** Questa *non* è una stringa, anche se sembra una. In realtà è un TextWriterFormat , che facoltativamente consente l'utilizzo di argomenti controllati in modo statico. Ma ai fini di un esempio di "ciao mondo" può essere pensato come una stringa.

F # Interactive

F # Interactive, è un ambiente REPL che ti consente di eseguire codice F #, una riga alla volta.

Se hai installato Visual Studio con F #, puoi eseguire F # Interactive nella console digitando `"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"` . Su Linux o OS X, il comando è invece `fsharpi` , che dovrebbe essere in `/usr/bin` o in `/usr/local/bin` seconda di come hai installato F # - in entrambi i casi, il comando dovrebbe essere sul tuo `PATH` modo da poter basta digitare `fsharpi` .

Esempio di utilizzo interattivo di 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
```

Usa `#help;;` per un aiuto

Si prega di notare l'uso di `;;` dire al REPL di eseguire qualsiasi comando precedentemente digitato.

Leggi Iniziare con F # online: https://riptutorial.com/it/fsharp/topic/817/iniziare-con-f-sharp

Capitolo 2: 1: codice WPF F # dietro l'applicazione con FsXaml

introduzione

La maggior parte degli esempi trovati per la programmazione F # WPF sembrano avere a che fare con il pattern MVVM, e alcuni con MVC, ma non c'è nessuno che mostri correttamente come rialzarsi e correre con il "buon vecchio" codice dietro.

Il codice dietro il pattern è molto facile da usare sia per l'insegnamento che per la sperimentazione. È utilizzato in numerosi libri introduttivi e materiale didattico sul web. Ecco perchè.

Questi esempi dimostreranno come creare un codice dietro l'applicazione con finestre, controlli, immagini e icone e altro ancora.

Examples

Creare un nuovo codice F # WPF dietro l'applicazione.

Creare un'applicazione console F #.

Cambia il **tipo** di **output** dell'applicazione nell'applicazione *Windows* .

Aggiungi il pacchetto **FsXaml** NuGet.

Aggiungi questi quattro file sorgente, nell'ordine elencato qui.

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.
```

Eliminare il file *Program.fs* dal progetto.

Modificare l' **azione** di *compilazione* in *Risorsa* per i due file xaml.

Aggiungi un riferimento all'assembly .NET **UIAutomationTypes** .

Compilare e correre.

Non è possibile utilizzare il designer per aggiungere gestori di eventi, ma non è affatto un problema. Basta aggiungerli manualmente nel codice sottostante, come si vede con i tre gestori in questo esempio, incluso il gestore per il pulsante di test.

AGGIORNAMENTO: è stato aggiunto a FsXaml un modo alternativo e probabilmente più elegante

per aggiungere gestori di eventi. Puoi aggiungere il gestore di eventi in XAML, come in C # ma devi farlo manualmente, quindi sostituire il membro corrispondente che si trova nel tuo tipo F #. Lo raccomando.

3: aggiungi un'icona a una finestra

È una buona idea conservare tutte le icone e le immagini in una o più cartelle.

Fare clic con il tasto destro del mouse sul progetto e utilizzare F # Power Tools / New Folder per creare una cartella denominata Immagini.

Su disco, inserisci l'icona nella nuova cartella *Immagini* .

Tornando in Visual Studio, fai clic con il tasto destro del mouse su *Immagini* e utilizza **Aggiungi / Esistente** , quindi mostra *Tutti i file (.*) ** per vedere il file dell'icona in modo che tu possa selezionarlo e quindi **aggiungerlo** .

Seleziona il file dell'icona e imposta la sua **azione Build** su *Risorsa* .

In MainWindow.xaml, usa l'attributo Icon come questo. Le linee circostanti sono mostrate per il contesto.

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

Prima di eseguire, fai una ricostruzione e non solo una costruzione. Questo perché Visual Studio non inserisce sempre il file dell'icona nell'eseguibile a meno che non si ricostruisca.

È la finestra, e non l'applicazione, che ora ha un'icona. Vedrai l'icona in alto a sinistra della finestra in fase di esecuzione e la vedrai nella barra delle applicazioni. Task Manager e Windows File Explorer non mostreranno questa icona, poiché visualizzano l'icona dell'applicazione anziché l'icona della finestra.

4: Aggiungi icona all'applicazione

Creare un file di testo denominato AppIcon.rc, con il seguente contenuto.

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

Avrai bisogno di un file di icone chiamato AppIcon.ico perché funzioni, ma ovviamente puoi modificare i nomi a tuo piacimento.

Esegui il seguente comando.

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

Se non riesci a trovare rc.exe in questa posizione, cercalo sotto **C: \ Programmi (x86) \ Kit di**

Windows . Se ancora non riesci a trovarlo, scarica Windows SDK da Microsoft.

Verrà generato un file denominato AppIcon.res.

In Visual Studio, apri le proprietà del progetto. Seleziona la pagina **dell'applicazione** .

Nella casella di testo intitolata **File di risorse** , digitare *AppIcon.res* (o *Images \ AppIcon.res* se lo si inserisce lì), quindi chiudere le proprietà del progetto da salvare.

Apparirà un messaggio di errore, affermando "Il file di risorse inserito non esiste Ignora questo messaggio di errore non riappare.

Ricostruire. L'eseguibile avrà quindi un'icona di applicazione e questo viene visualizzato in Esplora file. Durante l'esecuzione, questa icona apparirà anche in Task Manager.

2: aggiungi un controllo

Aggiungi questi due file in questo ordine sopra i file per la finestra principale.

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()
```

L' **azione** di *compilazione* per *MyControl.xaml* deve essere impostata su *Risorsa* .

Ovviamente, in seguito, dovrai aggiungere "come questo" nella dichiarazione di MyControl, esattamente come fatto per la finestra principale.

Nel file **MainWindow.xaml.fs** , nella classe per MainWindow, aggiungere questa riga

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

e aggiungi queste due linee nella sezione **do** della classe della finestra principale.

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

Non ci può essere più di un **fare** -sezione in una classe, e si rischia di averne bisogno quando si scrive un sacco di codice code-behind.

Al controllo è stato assegnato un colore di sfondo verde chiaro, in modo da poter vedere facilmente dove si trova.

Si noti che il controllo bloccherà il pulsante della finestra principale dalla vista. È oltre lo scopo di questi esempi insegnare il WPF in generale, quindi non lo risolveremo qui.

Come aggiungere controlli da librerie di terze parti

Se aggiungi controlli da librerie di terze parti in un progetto C # WPF, il file XAML avrà normalmente linee come questa.

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

Questo forse non funzionerà con FsXaml.

Il progettista e il compilatore accettano tale riga, ma probabilmente durante l'esecuzione si verificherà un'eccezione a causa del mancato rilevamento del tipo di terze parti durante la lettura di XAML.

Prova invece qualcosa come il seguente.

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

Questo quindi è un esempio di un controllo che dipende da quanto sopra.

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

La libreria utilizzata in questo esempio è Extended Wpf Toolkit, disponibile gratuitamente tramite NuGet o come programma di installazione. Se si scaricano le librerie tramite NuGet, i controlli non sono disponibili nella casella degli strumenti, ma vengono comunque visualizzati nella finestra di progettazione se li si aggiunge manualmente in XAML e le proprietà sono disponibili nel riquadro Proprietà.

Leggi 1: codice WPF F # dietro l'applicazione con FsXaml online:
https://riptutorial.com/it/fsharp/topic/9008/1--codice-wpf-f-sharp-dietro-l-applicazione-con-fsxaml

Capitolo 3: Classi

Examples

Dichiarazione di una classe

```
// 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
```

Creare un'istanza

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

Leggi Classi online: https://riptutorial.com/it/fsharp/topic/3003/classi

Capitolo 4: elenchi

Sintassi

- [] // una lista vuota.

  head :: tail // una cella di costruzione che contiene un elemento, una testa e una lista, coda. :: è chiamato l'operatore Cons.

  let list1 = [1; 2; 3] // Notare l'uso di un punto e virgola.

  let list2 = 0 :: list1 // result è [0; 1; 2; 3]

  let list3 = list1 @ list2 // result è [1; 2; 3; 0; 1; 2; 3]. @ è l'operatore append.

  let list4 = [1..3] // result è [1; 2; 3]

  let list5 = [1..2..10] // result è [1; 3; 5; 7; 9]

  let list6 = [for i in 1..10 do if i% 2 = 1 then yield i] // result is [1; 3; 5; 7; 9]

Examples

Uso di lista di base

```
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
```

Calcolo della somma totale di numeri in una lista

Per ricorsione

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

L'esempio sopra dice: "Guarda la `list` , è vuota? Return 0. Altrimenti è una lista non vuota, quindi potrebbe essere `[1]` , `[1; 2]` , [1; 2; 3] ecc. Se la `list` è [1], lega la `head` della variabile a `1` e `tail` a `[]` quindi esegui `head + sumTotal tail` .

Esempio di esecuzione:

```
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
```

Un modo più generale per incapsulare il modello sopra è usando le pieghe funzionali! `sumTotal` diventa questo:

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

Creare liste

Un modo per creare un elenco è posizionare gli elementi in due parentesi quadre, separate da punto e virgola. Gli elementi devono avere lo stesso tipo.

Esempio:

```
> 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]
```

Quando una lista non ha elemento, è vuota. Una lista vuota può essere dichiarata come segue:

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

Altro esempio

Per creare un elenco di byte, è sufficiente eseguire il cast degli interi:

```
> let bytes = [byte(55); byte(10); byte(100)];;
val bytes : byte list = [55uy; 10uy; 100uy]
```

È anche possibile definire liste di funzioni, di elementi di un tipo precedentemente definito, di oggetti di una classe, ecc.

Esempio

```
> 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]
```

Intervalli

Per alcuni tipi di elementi (int, float, char, ...), è possibile definire un elenco per l'elemento start e l'elemento end, utilizzando il modello seguente:

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

Esempi:

```
> 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]
```

Puoi anche specificare un passaggio per determinati tipi, con il seguente modello:

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

Esempi:

```
> 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]
```

Generatore

Un altro modo per creare una lista è di generarlo automaticamente usando il generatore.

Possiamo usare uno dei seguenti modelli:

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

o

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

Esempi

```
> 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]
```

operatori

Alcuni operatori possono essere usati per costruire liste:

Operatore Cons:

Questo operatore: è usato per aggiungere un elemento head ad una lista:

```
> 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]
```

Concatenazione

La concatenazione delle liste viene effettuata con l'operatore @.

```
> 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]
```

Leggi elenchi online: https://riptutorial.com/it/fsharp/topic/1268/elenchi

Capitolo 5: Estensioni di tipo e modulo

Osservazioni

In tutti i casi durante l'estensione di tipi e moduli, il codice di estensione deve essere aggiunto / caricato prima del codice che deve chiamarlo. Deve anche essere reso disponibile al codice chiamante aprendo / importando i namespace rilevanti.

Examples

Aggiunta di nuovi metodi / proprietà a tipi esistenti

F # consente di aggiungere funzioni come "membri" ai tipi quando vengono definiti (ad esempio, Tipi di record). Tuttavia, F # consente anche ai nuovi membri dell'istanza di essere aggiunti a tipi *esistenti* - anche quelli dichiarati altrove e in altre lingue .net.

L'esempio seguente aggiunge un nuovo metodo di istanza `Duplicate` a tutte le istanze di `String` .

```
type System.String with
    member this.Duplicate times =
        Array.init times (fun _ -> this)
```

Nota : `this` è un nome di variabile scelto arbitrariamente da usare per riferirsi all'istanza del tipo che viene esteso - `x` funzionerebbe altrettanto bene, ma forse sarebbe meno descrittivo.

Può quindi essere chiamato nei seguenti modi.

```
// 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
```

Questa funzionalità è molto simile ai Metodi di estensione in C #.

Le nuove proprietà possono anche essere aggiunte ai tipi esistenti allo stesso modo. Diventeranno automaticamente proprietà se il nuovo membro non accetta argomenti.

```
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
```

Aggiunta di nuove funzioni statiche ai tipi esistenti

F # consente di estendere i tipi esistenti con nuove funzioni statiche.

```
type System.String with
    static member EqualsCaseInsensitive (a, b) = String.Equals(a, b,
StringComparison.OrdinalIgnoreCase)
```

Questa nuova funzione può essere invocata in questo modo:

```
let x = String.EqualsCaseInsensitive("abc", "aBc")
// result is True
```

Questa caratteristica può significare che piuttosto che dover creare librerie "di utilità" di funzioni, possono essere aggiunte a tipi esistenti pertinenti. Questo può essere utile per creare più versioni compatibili con F # delle funzioni che consentono funzionalità come il currying .

```
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
```

Aggiunta di nuove funzioni a moduli e tipi esistenti tramite moduli

I moduli possono essere utilizzati per aggiungere nuove funzioni a moduli e tipi esistenti.

```
namespace FSharp.Collections

module List =
    let pair item1 item2 = [ item1; item2 ]
```

La nuova funzione può quindi essere chiamata come se fosse un membro originale di List.

```
open FSharp.Collections

module Testing =
    let result = List.pair "a" "b"
    // result is a list containing "a" and "b"
```

Leggi Estensioni di tipo e modulo online: https://riptutorial.com/it/fsharp/topic/2977/estensioni-di-tipo-e-modulo

Capitolo 6: F # su .NET Core

Examples

Creazione di un nuovo progetto tramite dotnet CLI

Dopo aver installato gli strumenti CLI .NET, puoi creare un nuovo progetto con il seguente comando:

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

Questo crea un programma a riga di comando.

Flusso di lavoro del progetto iniziale

Crea un nuovo progetto

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

Ripristina tutti i pacchetti elencati in project.json

```
dotnet restore
```

Dovrebbe essere scritto un file project.lock.json.

Esegui il programma

```
dotnet run
```

Quanto sopra compilerà il codice se necessario.

L'output del progetto predefinito creato da `dotnet new -lf#` contiene quanto segue:

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

Leggi F # su .NET Core online: https://riptutorial.com/it/fsharp/topic/4404/f-sharp-su--net-core

Capitolo 7: Flussi di lavoro di sequenza

Examples

resa e resa!

Nei flussi di lavoro in sequenza, la `yield` aggiunge un singolo elemento alla sequenza che si sta costruendo. (Nella terminologia monadica, è il `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!` (pronunciato *bang di rendimento*) inserisce tutti gli elementi di un'altra sequenza in questa sequenza in fase di costruzione. O, in altre parole, aggiunge una sequenza. (In relazione alle monadi, è `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']
```

Le sequenze create dai flussi di lavoro di sequenza sono anche pigre, il che significa che gli elementi della sequenza non vengono effettivamente valutati fino a quando non sono necessari. Alcuni modi per forzare gli elementi includono chiamare `Seq.take` (tira i primi n elementi in una sequenza), `Seq.iter` (applica una funzione a ciascun elemento per l'esecuzione di effetti collaterali), o `Seq.toList` (converte una sequenza in una lista) . Combinare questo con la ricorsione è dove `yield!` inizia davvero a brillare.

```
> 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]
```

per

`for` espressione in sequenza è progettato per assomigliare proprio al suo cugino più famoso, l'imperativo for-loop. "Scorre" attraverso una sequenza e valuta il corpo di ogni iterazione nella sequenza che sta generando. Proprio come tutte le sequenze correlate, NON è mutabile.

```
> 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]
```

Leggi Flussi di lavoro di sequenza online: https://riptutorial.com/it/fsharp/topic/2785/flussi-di-lavoro-di-sequenza

Capitolo 8: Folds

Examples

Introduzione alle pieghe, con una manciata di esempi

Le pieghe sono funzioni (di ordine superiore) utilizzate con sequenze di elementi. Crollano `seq<'a>` in `'b` dove `'b` è di qualsiasi tipo (eventualmente ancora `'a`). Questo è un po 'astratto, quindi lasciate entrare in esempi pratici concreti.

Calcolo della somma di tutti i numeri

In questo esempio, `'a` è un `int` . Abbiamo una lista di numeri e vogliamo calcolare sommarne tutti i numeri. Per sommare i numeri della lista `[1; 2; 3]` scriviamo

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

Lasciatemi spiegare, perché abbiamo a che fare con le liste, usiamo la `fold` nel modulo `List` , quindi `List.fold` . il primo argomento `fold` takes è una funzione binaria, la **cartella** . Il secondo argomento è il **valore iniziale** . `fold` inizia a piegare la lista applicando consecutivamente la funzione cartella agli elementi della lista iniziando dal valore iniziale e dal primo elemento. Se la lista è vuota, viene restituito il valore iniziale!

La panoramica schematica di un esempio di esecuzione si presenta così:

```
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
```

La funzione `List.sum` è approssimativamente `List.fold add LanguagePrimitives.GenericZero` dove lo zero generico lo rende compatibile con interi, float, interi grandi ecc.

Conteggio degli elemets in un elenco (`count` dell'implementazione)

Questo viene fatto quasi come sopra, ma ignorando il valore reale dell'elemento nella lista e aggiungendo invece 1.

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

Questo può anche essere fatto in questo modo:

```
[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
```

Quindi puoi definire il `count` come segue:

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

Trovare il massimo della lista

Questa volta useremo `List.reduce` che è come `List.fold` ma senza un valore iniziale come in questo caso in cui non sappiamo quale sia il tipo dei valori che stiamo confrontando:

```
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
```

Trovare il minimo di una lista

Proprio come quando si trova il massimo, la cartella è diversa

```
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
```

Elenchi concatenanti

Qui stiamo prendendo l'elenco delle liste La funzione cartella è l'operatore `@`

```
// [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]
```

Oppure potresti usare gli operatori binari come funzione della tua cartella:

```
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...
```

Calcolo del fattoriale di un numero

La stessa idea di quando si sommano i numeri, ma ora li moltiplichiamo. se vogliamo il fattoriale di `n` moltiplessiamo tutti gli elementi nella lista `[1 .. n]` . Il codice diventa:

```
// 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]
```

Implementare `forall` , `exists` e `contains`

la funzione `forall` controlla se tutti gli elementi di una sequenza soddisfano una condizione. `exists` controllo se almeno un elemento nell'elenco soddisfa la condizione. Per prima cosa dobbiamo sapere come comprimere un elenco di valori `bool` . Bene, usiamo le pieghe ofcourse! gli operatori booleani saranno le nostre funzioni di cartella.

Per verificare se tutti gli elementi in una lista sono `true` , li collassiamo con la funzione `&&` con `true` come valore iniziale.

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

Allo stesso modo, per verificare se un elemento è `true` in una lista di valori booleani, lo collassiamo con `||` operatore con `false` come valore iniziale:

```
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
```

Torna a `forall` ed `exists` . Qui prendiamo una lista di qualsiasi tipo, usiamo la condizione per trasformare tutti gli elementi in valori booleani e poi la comprimiamo:

```
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
```

Per verificare se tutti gli elementi in [1; 2; 3; 4] sono più piccoli di 5:

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

definire il metodo `contains` con `exists` :

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

O anche

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

Ora controlla se la lista [1 .. 5] contiene il valore 2:

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

Attuazione al `reverse` :

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

Implementazione di `map` e `filter`

```
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]
```

C'è qualcosa `fold` non si può fare? Non lo so davvero

Calcolo della somma di tutti gli elementi di una lista

Per calcolare la somma di termini (di tipo float, int o intero grande) di un elenco di numeri, è preferibile utilizzare List.sum In altri casi, List.fold è la funzione più adatta per calcolare tale somma.

1. Somma di numeri complessi

In questo esempio, dichiariamo un elenco di numeri complessi e calcoliamo la somma di tutti i termini nell'elenco.

All'inizio del programma, aggiungi un riferimento a System.Numerics

apri System.Numerics

Per calcolare la somma, inizializziamo l'accumulatore al numero complesso 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
```

Risultato:

```
(3, 51)
```

2. Somma dei numeri di tipo di unione

Supponiamo che una lista sia composta da numeri di tipo union (float o int) e voglia calcolare la somma di questi numeri.

Dichiarare prima del seguente tipo di numero:

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

Calcola la somma dei numeri del numero di tipo di una lista:

```
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
```

Risultato:

```
13.5
```

Il primo parametro della funzione, che rappresenta l'accumulatore, è di tipo float e il secondo parametro, che rappresenta un elemento nell'elenco, è di tipo numero. Ma prima di aggiungere, dobbiamo usare un pattern matching e cast per digitare float quando elem è di tipo Int.

Leggi Folds online: https://riptutorial.com/it/fsharp/topic/2250/folds

Capitolo 9: funzioni

Examples

Funzioni di più di un parametro

In F #, **tutte le funzioni richiedono esattamente un parametro** . Questa sembra un'affermazione strana, dal momento che è banalmente facile dichiarare più di un parametro in una dichiarazione di funzione:

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

Ma se si digita la dichiarazione di funzione nell'interprete interattivo F #, vedrete che la sua firma di tipo è:

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

Senza i nomi dei parametri, quella firma è `int -> int -> int` . L'operatore `->` è right-associative, il che significa che questa firma è equivalente a `int -> (int -> int)` . In altre parole, `add` è una funzione che accetta un parametro `int` e restituisce **una funzione che accetta un `int` e restituisce `int`** . Provalo:

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

Tuttavia, puoi anche chiamare una funzione come `add` in un modo più "convenzionale", passandogli direttamente due parametri, e funzionerà come ti aspetteresti:

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

Questo vale per le funzioni con tutti i parametri che vuoi:

```
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
```

Questo metodo di pensare alle funzioni multiparametro è come funzioni che accettano un parametro e restituiscono nuove funzioni (che a loro volta possono prendere un parametro e

restituire nuove funzioni, finché non si raggiunge la funzione finale che prende il parametro finale e infine restituisce un risultato) Si chiama **curry** , in onore del matematico Haskell Curry, che è famoso per lo sviluppo del concetto. (È stato inventato da qualcun altro, ma Curry merita meritatamente la maggior parte del merito.)

Questo concetto è usato in F # e vorrete avere familiarità con esso.

Nozioni di base sulle funzioni

La maggior parte delle funzioni in F # vengono create con la sintassi `let` :

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

Definisce una funzione denominata `timesTwo` che accetta un singolo parametro `x` . Se si esegue una sessione F # interattiva (`fsharpi` su OS X e Linux, `fsi.exe` su Windows) e si incolla la funzione in (e si aggiunge il `;;` che dice a `fsharpi` di valutare il codice appena digitato), vedrai che risponde con:

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

Ciò significa che `timesTwo` è una funzione che accetta un singolo parametro `x` di tipo `int` e restituisce un `int` . Le firme delle funzioni vengono spesso scritte senza i nomi dei parametri, quindi vedrai spesso questo tipo di funzione scritto come `int -> int` .

Ma aspetta! In che modo F # sapeva che `x` era un parametro intero, dal momento che non hai mai specificato il suo tipo? Questo è dovuto al **tipo di inferenza** . Perché nel corpo della funzione, hai moltiplicato `x` per `2` , i tipi di `x` e `2` devono essere uguali. (Come regola generale, F # non invierà implicitamente i valori a tipi diversi, ma specificherà esplicitamente qualsiasi tipo di digitazione che si desidera).

Se vuoi creare una funzione che non assuma alcun parametro, questo è il modo **sbagliato** per farlo:

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

Il modo **giusto** per farlo è:

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

Questa funzione `hello` ha l' `unit -> unit` , che è spiegata nel tipo "unità" .

Funzioni al curry vs tupla

Esistono due modi per definire le funzioni con più parametri in F #, funzioni Curried e funzioni Tupled.

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

Tutte le funzioni definite da F # esterne (come il framework .NET) vengono utilizzate in F # con il modulo Tupled. La maggior parte delle funzioni nei moduli di base F # vengono utilizzate con il modulo Curried.

La forma Curried è considerata F # idiomatica, perché consente un'applicazione parziale. Nessuno dei seguenti due esempi è possibile con la forma Tupla.

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

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

La ragione è che la funzione Curried, quando viene chiamata con un parametro, restituisce una funzione. Benvenuto nella programmazione funzionale !!

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

Puoi vedere che è esattamente la stessa firma.

Tuttavia, quando si interfaccia con un altro codice .NET, come quando si scrivono le librerie, è importante definire le funzioni usando il modulo Tupla.

inlining

Inlining consente di sostituire una chiamata a una funzione con il corpo della funzione.

Questo a volte è utile per motivi di prestazioni su parte critica del codice. Ma la controparte è che il vostro assembly richiederà molto spazio poiché il corpo della funzione è duplicato ovunque si sia verificata una chiamata. Devi stare attento quando decidi se integrare una funzione o meno.

Una funzione può essere sottolineata con la parola chiave `inline` :

```
// 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"
```

Un altro esempio con valore locale:

```
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
```

Tubo avanti e indietro

Gli operatori di tubi sono utilizzati per passare parametri a una funzione in modo semplice ed elegante. Permette di eliminare i valori intermedi e facilitare la lettura delle chiamate di funzione.

In F # ci sono due operatori di pipe:

- **Inoltra** (`|>`): passaggio dei parametri da sinistra a destra

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

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

- **Indietro** (`<|`): passaggio dei parametri da destra a sinistra

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

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

Ecco un esempio senza operatori di pipe:

```
// 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
```

Possiamo abbreviare l'esempio precedente e renderlo più pulito con l'operatore pipe in avanti:

```
// 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
```

Ogni risultato della funzione viene passato come parametro alla funzione successiva.

Se si desidera passare più parametri all'operatore del tubo, è necessario aggiungere un `|` per ogni parametro aggiuntivo e creare una Tupla con i parametri. L'operatore di pipe Native F # supporta fino a tre parametri (|||> o <|||).

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

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

Leggi funzioni online: https://riptutorial.com/it/fsharp/topic/2525/funzioni

Capitolo 10: Generics

Examples

Inversione di un elenco di qualsiasi tipo

Per invertire una lista, non è importante che tipo siano gli elementi della lista, solo in che ordine si trovano. Questo è un candidato perfetto per una funzione generica, quindi la stessa funzione di reverseal può essere usata indipendentemente da quale lista viene passata.

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

Il codice non fa ipotesi sui tipi di elementi. Il compilatore (o F # interattivo) ti dirà che la firma del tipo di questa funzione è `'T list -> 'T list` Il `'T` ti dice che è un tipo generico senza vincoli. Puoi anche vedere `'a` anziché `'T` - la lettera non è importante perché è solo un segnaposto *generico* . Possiamo passare un `int list` o una `string list` , ed entrambe funzioneranno correttamente, restituendo rispettivamente un `int list` o una `string list` .

Ad esempio, in F # interattivo:

```
> 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"]
```

Mappatura di un elenco in un tipo diverso

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

La firma di questa funzione è `('a -> 'b) -> 'a list -> 'b list` , che è il più generico possibile. Ciò non impedisce ad `'a` di essere dello stesso tipo di essere `'b` , ma permette anche che siano diversi. Qui puoi vedere che `'a` tipo che è il parametro della funzione `f` deve corrispondere al tipo del parametro `list` . Questa funzione è ancora generica, ma ci sono alcuni lievi vincoli sugli input - se i tipi non corrispondono, ci sarà un errore di compilazione.

Esempi:

```
> 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'
```

Leggi Generics online: https://riptutorial.com/it/fsharp/topic/7731/generics

Capitolo 11: Il tipo "unità"

Examples

A che serve una tupla 0?

Una tupla di 2 o una tupla di 3 rappresentano un gruppo di elementi correlati. (Punti nello spazio 2D, valori RGB di un colore, ecc.) Una tupla 1 non è molto utile in quanto potrebbe essere facilmente sostituita con un singolo `int` .

Una tupla 0 sembra ancora più inutile poiché non contiene assolutamente *nulla* . Tuttavia ha proprietà che lo rendono molto utile in linguaggi funzionali come F #. Ad esempio, il tipo di tupla 0 ha esattamente *un* valore, solitamente rappresentato come `()` . Tutte le tuple 0 hanno questo valore quindi è essenzialmente un tipo singleton. Nella maggior parte dei linguaggi di programmazione funzionale, incluso F #, questo è chiamato il tipo di `unit` .

Le funzioni che restituiscono `void` in C # restituiranno il tipo di `unit` in F #:

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

Eseguilo nell'interprete interattivo F # e vedrai:

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

Ciò significa che il valore `printResult` è di tipo `unit` e ha il valore `()` (la tupla vuota, l'unico valore del tipo di `unit`).

Le funzioni possono anche prendere il tipo di `unit` come parametro. In F #, le funzioni potrebbero sembrare che non stanno prendendo alcun parametro. Ma in effetti stanno prendendo un singolo parametro `unit` di tipo. Questa funzione:

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

è in realtà equivalente a:

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

Cioè, una funzione che accetta un parametro di tipo `unit` e restituisce il valore `int` 6. Se si osserva la firma del tipo che l'interprete interattivo F # stampa quando si definisce questa funzione, vedrete:

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

Il fatto che tutte le funzioni richiedono almeno un parametro e restituiscono un valore, anche se tale valore è talvolta un valore "inutile" come `()` , significa che la composizione della funzione è

molto più semplice in F # rispetto alle lingue che non hanno il tipo di `unit` . Ma questo è un argomento più avanzato che vedremo più avanti. Per ora, ricorda che quando vedi l' `unit` in una firma di funzione, o `()` nei parametri di una funzione, questo è il tipo di tupla 0 che funge da modo per dire "Questa funzione accetta o restituisce valori privi di significato".

Differire nell'esecuzione del codice

Possiamo usare il tipo di `unit` come argomento di funzione per definire funzioni che non vogliamo vengano eseguite fino a più tardi. Questo è spesso utile in attività di background asincrone, quando il thread principale potrebbe voler attivare alcune funzionalità predefinite del thread in background, come magari spostarlo in un nuovo file, o se aa let-binding non dovrebbe essere eseguito immediatamente:

```
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)
```

Nel codice seguente, definiamo il codice per avviare un "worker" che stampa semplicemente il valore su cui sta lavorando ogni 2 secondi. L'operatore restituisce quindi due funzioni che possono essere utilizzate per controllarlo, una che la sposta al valore successivo su cui lavorare e una che ne impedisce il funzionamento. Queste devono essere funzioni, perché non vogliamo che i loro corpi siano eseguiti fino a quando non lo decidiamo, altrimenti il lavoratore si sposterebbe immediatamente al secondo valore e si spegnerebbe senza aver fatto nulla.

```
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
```

Possiamo quindi iniziare un lavoratore facendo

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

e il lavoro inizierà - se siamo in F # interattiva, vedremo i messaggi stampati nella console ogni due secondi. Possiamo quindi correre

```
nextValue ()
```

e vedremo i messaggi che indicano che il valore su cui si sta lavorando è passato a quello

successivo.

Quando è il momento di terminare il lavoro, possiamo eseguire il

```
stopOnNextTick ()
```

funzione, che stamperà il messaggio di chiusura, quindi uscirà.

Il tipo di `unit` è importante qui per indicare "nessun input": le funzioni dispongono già di tutte le informazioni necessarie per lavorare su di esse e il chiamante non può modificarlo.

Leggi Il tipo "unità" online: https://riptutorial.com/it/fsharp/topic/2513/il-tipo--unita-

Capitolo 12: Implementazione del modello di progettazione in F

Examples

Programmazione basata sui dati in F

Grazie all'inferenza di tipo e all'applicazione parziale nella programmazione `F#` data-driven è succinta e leggibile.

Immaginiamo che stiamo vendendo un'assicurazione auto. Prima di provare a venderlo a un cliente, proviamo a determinare se il cliente è un potenziale cliente valido per la nostra azienda controllando il sesso e l'età del cliente.

Un semplice modello cliente:

```
type Sex =
  | Male
  | Female

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

Successivamente vogliamo definire un elenco di esclusione (tabella) in modo che se un cliente corrisponde a qualsiasi riga nell'elenco di esclusione, il cliente non può acquistare la nostra assicurazione auto.

```
//  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
  |]
```

A causa dell'inferenza di tipo e dell'applicazione parziale, l'elenco di esclusione è flessibile ma facile da comprendere.

Infine, definiamo una funzione che utilizza l'elenco di esclusione (una tabella) per dividere i clienti in due bucket: clienti potenziali e negati.

```
// 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 ()
```

Per concludere, definiamo alcuni clienti e vediamo se sono potenziali clienti per la nostra assicurazione auto tra di loro:

```
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
```

Questo stampa:

```
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;})|]
```

Leggi Implementazione del modello di progettazione in F # online:
https://riptutorial.com/it/fsharp/topic/3925/implementazione-del-modello-di-progettazione-in-f-sharp

Capitolo 13: Introduzione a WPF in F

introduzione

Questo argomento illustra come sfruttare la **programmazione funzionale** in **un'applicazione WPF** . Il primo esempio proviene da un post di Māris Krivtežs (sezione *Remarks* ref in basso). La ragione per rivisitare questo progetto è duplice:

1 \ Il design supporta la separazione delle preoccupazioni, mentre il modello è mantenuto puro e le modifiche sono propagate in modo funzionale.

2 \ La somiglianza faciliterà la transizione all'implementazione di Gjallarhorn.

Osservazioni

Progetti demo della biblioteca @ GitHub

- FSharp.ViewModule (sotto FsXaml)
- Gjallarhorn (campioni di riferimento)

Māris Krivtežs ha scritto due fantastici post su questo argomento:

- Applicazione F # XAML - MVVM vs MVC in cui vengono evidenziati i pro e i contro di entrambi gli approcci.

  Ritengo che nessuno di questi stili di applicazione XAML tragga un grande beneficio dalla programmazione funzionale. Immagino che l'applicazione ideale consista nella vista che produce eventi ed eventi che mantengono lo stato attuale della vista. Tutta la logica applicativa dovrebbe essere gestita filtrando e manipolando eventi e modelli di vista, e nell'output dovrebbe produrre un nuovo modello di vista che è legato alla vista.

- F # XAML - MVVM basato sugli eventi come rivisitato nell'argomento sopra.

Examples

FSharp.ViewModule

La nostra app demo è composta da un tabellone. Il modello del punteggio è un record immutabile. Gli eventi segnapunti sono contenuti in un Tipo Unione.

```
namespace Score.Model

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

Le modifiche vengono propagate ascoltando gli eventi e aggiornando il modello di visualizzazione

di conseguenza. Invece di aggiungere membri al tipo di modello, come in OOP, dichiariamo un modulo separato per ospitare le operazioni consentite.

```
[<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
```

Il nostro modello di visualizzazione deriva da `EventViewModelBase<'a>` , che ha una proprietà `EventStream` di tipo `IObservable<'a>` . In questo caso gli eventi a cui vogliamo iscriversi sono di tipo `ScoringEvent` .

Il controller gestisce gli eventi in modo funzionale. Il suo `Score -> ScoringEvent -> Score` firma `Score -> ScoringEvent -> Score` ci mostra che, ogni volta che si verifica un evento, il valore corrente del modello viene trasformato in un nuovo valore. Ciò consente al nostro modello di rimanere puro, sebbene il nostro modello di vista non lo sia.

Un `eventHandler` compito di modificare lo stato della vista. Ereditando da `EventViewModelBase<'a>` possiamo usare `EventValueCommand` ed `EventValueCommandChecked` per collegare gli eventi ai comandi.

```
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
```

Il codice dietro il file (* .xaml.fs) è dove tutto è messo insieme, cioè la funzione di aggiornamento (`controller`) viene iniettata nel `MainViewModel` .

```
namespace Score.Views

open FsXaml

type MainView = XAML<"MainWindow.xaml">

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

Il tipo `CompositionRoot` funge da wrapper a cui viene fatto riferimento nel file XAML.

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

Non approfondirò più nel file XAML dato che è roba di base di WPF, l'intero progetto può essere trovato su GitHub .

Gjallarhorn

I tipi fondamentali nella biblioteca Gjallarhorn implementano `IObservable<'a>` , che renderà l'attuazione aspetto familiare (ricordate `EventStream` proprietà dall'esempio FSharp.ViewModule). L'unica vera modifica al nostro modello è l'ordine degli argomenti della funzione di aggiornamento. Inoltre, ora usiamo il termine *Messaggio* anziché *Evento* .

```
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
```

Per costruire un'interfaccia utente con Gjallarhorn, invece di creare classi per supportare l'associazione dati, creiamo semplici funzioni denominate `Component` . Nel loro costruttore la prima `source` argomenti è di tipo `BindingSource` (definita in Gjallarhorn.Bindable) e utilizzata per associare il modello alla vista e gli eventi dalla vista ai messaggi.

```
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
        ]
```

L'implementazione corrente differisce dalla versione di FSharp.ViewModule in quanto due comandi non hanno ancora correttamente implementato CanExecute. Elenca anche l'impianto idraulico dell'applicazione.

```
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
```

A sinistra con l'impostazione della vista disaccoppiata, combinando il tipo `MainWindow` e l'applicazione logica.

```
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
```

Questo riassume i concetti fondamentali, per ulteriori informazioni e un esempio più elaborato si prega di fare riferimento al post di Reed Copsey . Il progetto *Alberi di Natale* evidenzia un paio di vantaggi a questo approccio:

- Ci riscattiamo efficacemente dalla necessità di copiare (manualmente) il modello in una collezione personalizzata di modelli di vista, gestendoli e ricostruendo manualmente il modello dai risultati.
- Gli aggiornamenti all'interno delle raccolte vengono eseguiti in modo trasparente, pur mantenendo un modello puro.
- La logica e la vista sono ospitate da due diversi progetti, sottolineando la separazione delle preoccupazioni.

Leggi Introduzione a WPF in F # online: https://riptutorial.com/it/fsharp/topic/8758/introduzione-a-wpf-in-f-sharp

Capitolo 14: Memoizzazione

Examples

Memoizzazione semplice

La Memoizzazione consiste nei risultati della funzione di memorizzazione nella cache per evitare di calcolare lo stesso risultato più volte. Ciò è utile quando si lavora con funzioni che eseguono calcoli costosi.

Possiamo usare una semplice funzione fattoriale come esempio:

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

    innerLoop index 1
```

Chiamando questa funzione più volte con lo stesso parametro si ottiene sempre lo stesso calcolo.

La Memoizzazione ci aiuterà a memorizzare nella cache il risultato fattoriale e a restituirlo se lo stesso parametro viene passato di nuovo.

Ecco una semplice implementazione di memoization:

```
// 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
```

La funzione di `memoization` prende semplicemente una funzione come parametro e restituisce una funzione con la stessa firma. Si può vedere questo nel metodo signature `f:('a -> 'b) -> ('a -> 'b)` . In questo modo puoi usare la memoizzazione come se stessi chiamando il metodo fattoriale.

Le chiamate `printfn` servono a mostrare cosa succede quando chiamiamo la funzione più volte; possono essere rimossi in modo sicuro.

Usare la memoizzazione è facile:

```
// 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
```

Memoizzazione in una funzione ricorsiva

Usando l'esempio precedente di calcolare il fattoriale di un intero, inserire nella tabella hash tutti i valori fattoriali calcolati all'interno della ricorsione, che non appaiono nella tabella.

Come nell'articolo sulla memoizzazione , dichiariamo una funzione `f` che accetta un `fact` parametro funzione e un parametro intero. Questa funzione `f` include istruzioni per calcolare il fattoriale di `n` da `fact(n-1)` .

Ciò consente di gestire la ricorsione mediante la funzione restituita da `memorec` e non dal `fact` stesso ed eventualmente interrompere il calcolo se il valore dei `fact(n-1)` è già stato calcolato e si trova nella tabella hash.

```
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
```

Risultato:

```
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
```

Leggi Memoizzazione online: https://riptutorial.com/it/fsharp/topic/2698/memoizzazione

Capitolo 15: Modelli attivi

Examples

Modelli attivi semplici

I pattern attivi sono un tipo speciale di pattern matching in cui è possibile specificare le categorie denominate in cui i dati potrebbero cadere e quindi utilizzare tali categorie nelle dichiarazioni `match` .

Per definire un modello attivo che classifica i numeri come positivi, negativi o pari a zero:

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

Questo può quindi essere utilizzato in un'espressione di corrispondenza del modello:

```
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
```

Pattern attivi con parametri

Gli schemi attivi sono solo semplici funzioni.

Come le funzioni puoi definire parametri aggiuntivi:

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

Questo può essere utilizzato in un modello che corrisponde in questo modo:

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

I pattern attivi possono essere utilizzati per convalidare e trasformare gli

argomenti delle funzioni

Un uso interessante ma piuttosto sconosciuto di Active Pattern in `F#` è che possono essere usati per validare e trasformare gli argomenti delle funzioni.

Considera il modo classico di convalidare argomenti:

```
// 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 ()
```

In genere aggiungiamo codice nel metodo per verificare che gli argomenti siano corretti. Usando Pattern attivi in `F#` possiamo generalizzare questo e dichiarare l'intenzione nella dichiarazione di argomento.

Il seguente codice è equivalente al codice sopra:

```
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 ()
```

Per l'utente della funzione `f` e `g` non c'è differenza tra le due versioni differenti.

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

Un problema è se i Pattern attivi aggiungono un sovraccarico alle prestazioni. Usiamo `ILSpy` per decompilare `f` e `g` per vedere se è così.

```
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();
}
```

Grazie a `inline` Active Patterns non aggiunge extra overhead rispetto al classico modo di convalidare l'argomento doing.

Pattern attivi come wrapper API .NET

I pattern attivi possono essere utilizzati per rendere più naturale la chiamata ad alcune API .NET, in particolare quelle che utilizzano un parametro di output per restituire più del semplice valore restituito dalla funzione.

Ad esempio, normalmente chiameresti il metodo `System.Int32.TryParse` come segue:

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

Puoi migliorarlo un po 'usando la corrispondenza del modello:

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

Tuttavia, possiamo anche definire il seguente pattern attivo che avvolge la funzione `System.Int32.TryParse` :

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

Ora possiamo fare quanto segue:

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

Un altro buon candidato per essere inclusi in un Pattern attivo sono le API delle espressioni regolari:

```
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"
```

Modelli attivi completi e parziali

Esistono due tipi di Pattern attivi che differiscono in qualche modo nell'utilizzo: Completo e Parziale.

I modelli attivi completi possono essere utilizzati quando è possibile enumerare tutti i risultati, ad esempio "è un numero pari o dispari?"

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

Si noti che la definizione del modello attivo elenca entrambi i casi possibili e nient'altro, e il corpo restituisce uno dei casi elencati. Quando lo usi in un'espressione di corrispondenza, questa è una corrispondenza completa:

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

Questo è utile quando si desidera suddividere lo spazio di input in categorie note che lo coprano completamente.

D'altra parte, i pattern attivi parziali consentono di ignorare esplicitamente alcuni risultati restituendo `option` . La loro definizione utilizza un caso speciale di _ per il caso ineguagliato.

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

In questo modo possiamo eguagliare anche quando alcuni casi non possono essere gestiti dalla nostra funzione di analisi.

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

I Pattern attivi parziali possono essere utilizzati come forma di test, indipendentemente dal fatto

che l'input cada in una categoria specifica nello spazio di input ignorando le altre opzioni.

Leggi Modelli attivi online: https://riptutorial.com/it/fsharp/topic/962/modelli-attivi

Capitolo 16: monadi

Examples

Comprendere le Monadi deriva dalla pratica

Questo argomento è destinato agli sviluppatori di F # da intermedio ad avanzato

"Cosa sono le Monadi?" è una domanda comune È facile rispondere, ma come nella guida di Hitchhikers alla galassia ci rendiamo conto che non capiamo la risposta perché non sapevamo che cosa stavamo chiedendo.

Molti credono che il modo di comprendere la Monade sia praticarli. Come programmatori di solito non ci interessa il fondamento matematico per il Principio di sostituzione di Liskov, i sottotipi o le sottoclassi. Usando queste idee abbiamo acquisito un'intuizione per ciò che rappresentano. Lo stesso è vero per Monads.

Per aiutarti a iniziare con Monads questo esempio dimostra come costruire una libreria Monadic Parser Combinator . Questo potrebbe aiutarti a iniziare ma la comprensione verrà dalla scrittura della tua libreria Monadic.

Basta prosa, tempo per il codice

Il tipo di parser:

```
// 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)
```

Usando questa definizione di un parser definiamo alcune funzioni fondamentali del parser

```
// 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` è una funzione che data una funzione `sat` produce un parser che ha successo se non abbiamo passato `EOS` e il carattere nella posizione corrente passa la funzione `sat` . Usando `satisfy` creiamo un numero di parser di caratteri utili.

Eseguendo questo 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)
```

Abbiamo alcuni parser fondamentali in atto. Li combineremo in parser più potenti usando le funzioni del combinatore di parser

```
// '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)
```

I nomi e le firme non sono scelti arbitrariamente, ma non approfondiremo questo argomento, vediamo invece come utilizziamo il `bind` per combinare il parser in quelli più complessi:

```
> 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)
```

Ciò che questo ci mostra è che il `bind` ci consente di combinare due parser in un parser più complesso. Come risultato del `bind` è un parser che a sua volta può essere combinato di nuovo.

```
> 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` sarà il modo fondamentale in cui combiniamo i parser, anche se definiremo le funzioni di supporto per semplificare la sintassi.

Una delle cose che può semplificare la sintassi sono espressioni di calcolo . Sono facili da definire:

```
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)
```

Questo è equivalente a:

```
> 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)
```

Un altro combinatore di parser fondamentale che useremo alot è `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
```

Questo ci permette di definire `letterOrDigit` questo modo:

```
> 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)
```

Una nota sugli operatori Infix

Una preoccupazione comune su FP è l'uso di operatori inusuali insoliti come `>>=` , `>=>` , `<-` e così via. Tuttavia, la maggior parte non sono interessati all'uso di `+` , `-` , `*` , `/` e `%` , questi sono operatori ben noti usati per comporre valori. Tuttavia, una parte importante della FP riguarda la composizione non solo dei valori ma anche delle funzioni. Per uno sviluppatore FP intermedio gli operatori infissi `>>=` , `>=>` , `<-` sono ben noti e dovrebbero avere firme specifiche e semantica.

Per le funzioni che abbiamo definito finora dovremmo definire i seguenti operatori infissi usati per combinare i parser:

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

Quindi `>>=` significa `bind` e `<|>` significa `orElse` .

Questo ci consente di combinare parser più succinti:

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

Per definire alcuni combinatori di parser avanzati che ci consentano di analizzare espressioni più complesse, definiamo alcuni più semplici combinatori di parser:

```
// '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
  }
```

Siamo pronti a definire `many` e `sepBy` che sono più avanzati in quanto applicano i parser di input finché non falliscono. Quindi `many` e `sepBy` restituiscono il risultato aggregato:

```
// '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
```

Creazione di un parser di espressioni semplici

Con gli strumenti che abbiamo creato possiamo ora definire un parser per espressioni semplici come `1+2*3`

Iniziamo dal basso definendo un parser per la `pint` interi

```
// '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)
  }
```

Cerchiamo di analizzare il maggior numero di cifre possibile, il risultato è una `char list` . Se la lista è vuota `fail` , altrimenti pieghiamo i caratteri in un numero intero.

Test di `pint` in FSI:

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

Inoltre, è necessario analizzare il diverso tipo di operatori utilizzati per combinare valori interi:

```
// 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)
```

Legando tutto insieme:

```
// '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
  }
```

Eseguendo tutto in FSI:

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

Conclusione

Definendo `Parser<'T>` , `return_` , `bind` e assicurandosi che obbediscano alle leggi monadiche abbiamo costruito un semplice ma potente framework Monadic Parser Combinator.

Monadi e parser vanno insieme perché i parser vengono eseguiti su uno stato parser. Monade consente di combinare parser nascondendo allo stesso tempo lo stato del parser, riducendo il disordine e migliorando la componibilità.

Il framework che abbiamo creato è lento e non produce messaggi di errore, questo per mantenere il codice succinto. FParsec fornisce sia prestazioni accettabili che eccellenti messaggi di errore.

Tuttavia, un esempio da solo non può dare la comprensione di Monadi. Bisogna praticare le monadi.

Ecco alcuni esempi su Monade che puoi provare ad implementare per raggiungere la tua comprensione vincente:

1. Monade di stato: consente lo svolgimento di uno stato di ambiente nascosto implicitamente
2. Tracer Monad - Permette di trasportare lo stato della traccia in modo implicito. Una variante della Monade di Stato
3. Turtle Monad - A Monad per la creazione di programmi Turtle (Logos). Una variante della Monade di Stato
4. Continuazione Monad - A coroutine Monad. Un esempio di questo è `async` in F #

La cosa migliore per imparare sarebbe trovare un'applicazione per Monade in un dominio con cui ti trovi bene. Per me quello era parser.

Codice sorgente completo:

```
// 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
  }
```

Le espressioni di calcolo forniscono una sintassi alternativa per collegare le Monade

Relativi ai Monadi sono le espressioni di calcolo `F#` (`CE`). Un programmatore di solito implementa un `CE` per fornire un approccio alternativo al concatenamento delle Monade, ovvero invece di questo:

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

Puoi scrivere questo:

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

Entrambi gli stili sono equivalenti e spetta alle preferenze degli sviluppatori quali scegliere.

Per dimostrare come implementare una `CE` immagina che ti piacciano tutte le tracce per includere un id di correlazione. Questo ID di correlazione aiuterà a correlare le tracce che appartengono alla stessa chiamata. Ciò è molto utile quando i file di registro contengono tracce provenienti da chiamate simultanee.

Il problema è che è complicato includere l'id di correlazione come argomento per tutte le funzioni. Poiché Monads consente di trasportare lo stato implicito , definiremo una Log Monad per nascondere il contesto del log (cioè l'id di correlazione).

Iniziamo definendo un contesto di log e il tipo di una funzione che traccia con il contesto del log:

```
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 ())
```

Definiamo anche due funzioni di traccia che registreranno con l'ID di correlazione dal contesto del registro:

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

`trace` è una `Function<unit>` che significa che verrà passato un contesto di log quando invocato. Dal contesto del registro, prendiamo l'id di correlazione e lo tracciamo insieme a `v`

Inoltre definiamo `bind` e `return_` e mentre seguono il leggi Monade questa forma il nostro Log Monade.

```
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
```

Definiamo infine `LogBuilder` che ci consentirà di utilizzare la sintassi `CE` per concatenare le Monade `Log` .

```
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 ()
```

Ora possiamo definire le nostre funzioni che dovrebbero avere il contesto implicito del registro:

```
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
  }
```

Eseguiamo g con:

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

Quale stampa:

```
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
```

Si noti che CorrelationId viene trasportato implicitamente da `run` a `g` to `f` che ci consente di correlare le voci del log durante la risoluzione dei problemi.

`CE` ha molte più funzioni ma questo dovrebbe aiutarti a iniziare a definire il tuo `CE` : s.

Codice completo:

```
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
```

Leggi monadi online: https://riptutorial.com/it/fsharp/topic/3320/monadi

Capitolo 17: operatori

Examples

Come comporre valori e funzioni utilizzando operatori comuni

Nella programmazione orientata agli oggetti un compito comune è comporre oggetti (valori). Nella programmazione funzionale è compito comune comporre valori e funzioni.

Siamo abituati a comporre valori dalla nostra esperienza di altri linguaggi di programmazione usando operatori come `+` , `-` , `*` , `/` e così via.

Composizione del valore

```
let x = 1 + 2 + 3 * 2
```

Poiché la programmazione funzionale compone funzioni e valori, non sorprende che esistano operatori comuni per la composizione di funzioni come `>>` , `<<` , `|>` e `<|` .

Composizione funzionale

```
// 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#` tubazioni in avanti sono preferite rispetto alle tubazioni inverse perché:

1. Digitare un'inferenza (generalmente) scorre da sinistra a destra, quindi è naturale che valori e funzioni scorrano anche da sinistra a destra
2. Perché `<|` e `<<` dovrebbe avere la giusta associatività, ma in `F#` sono lasciati associativi che ci costringono a inserire ()
3. La miscelazione delle tubazioni avanti e indietro generalmente non funziona perché hanno la stessa precedenza.

Composizione di Monad

Dato che Monads (come `Option<'T>` o `List<'T>`) sono comunemente usati nella programmazione funzionale ci sono anche operatori comuni ma meno conosciuti per comporre funzioni che

funzionano con Monads come `>>=` , `>=>` , `<|>` e `<*>` .

```
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
```

Conclusione

Per i nuovi programmatori funzionali la composizione delle funzioni utilizzando gli operatori potrebbe sembrare opaca e oscura, ma è perché il significato di questi operatori non è comunemente noto come operatori che lavorano sui valori. Tuttavia, con un po 'di allenamento usando `|>` , `>>` , `>>=` e `>=>` diventa naturale come usare `+` , `-` , `*` e `/` .

Latebinding in F # usando? operatore

In un linguaggio tipizzato staticamente come `F#` lavoriamo con tipi ben noti in fase di compilazione. Consumiamo fonti di dati esterne in un modo sicuro per tipo utilizzando i provider di tipi.

Tuttavia, occasionalmente è necessario utilizzare l'associazione tardiva (come la `dynamic` in `C#`). Ad esempio quando si lavora con documenti `JSON` che non hanno uno schema ben definito.

Per semplificare il lavoro con l'associazione tardiva, `F#` fornisce supporto agli operatori di ricerca dinamica `?` e `?<-` .

Esempio:

```
// (?) 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
```

Si scopre che il supporto `F#` per l'associazione tardiva è semplice ma flessibile.

Leggi operatori online: https://riptutorial.com/it/fsharp/topic/4641/operatori

Capitolo 18: Parametri di tipo staticamente risolti

Sintassi

- `s` è un'istanza di `^a` si desidera accettare in fase di compilazione, che può essere qualsiasi cosa che implementa i membri effettivamente chiamati usando la sintassi.
- `^a` è simile ai generici che sarebbero `'a` (o `'A` o `'T` per esempio) ma questi sono risolti in tempo di compilazione, e permettono tutto ciò che si adatta a tutti gli usi richiesti all'interno del metodo. (nessuna interfaccia richiesta)

Examples

Utilizzo semplice per tutto ciò che ha un membro di lunghezza

```
let inline getLength s = (^a: (member Length: _) s)
//usage:
getLength "Hello World" // or "Hello World" |> getLength
// returns 11
```

Classe, interfaccia, utilizzo del record

```
// 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)
```

Chiamata membro statico

questo accetterà qualsiasi tipo con un metodo chiamato `GetLength` che non accetta nulla e restituisce un int:

```
((^a : (static member GetLength : int) ()))
```

Leggi Parametri di tipo staticamente risolti online:

https://riptutorial.com/it/fsharp/topic/7228/parametri-di-tipo-staticamente-risolti

Capitolo 19: Pattern Matching

Osservazioni

Pattern Matching è una potente funzionalità di molti linguaggi funzionali in quanto spesso consente di gestire la ramificazione in modo molto succinto rispetto all'utilizzo di più istruzioni di stile `if` / `else if` / `else` . Tuttavia, date abbastanza opzioni e "quando" le guardie , Pattern Matching può anche diventare prolisso e difficile da capire a colpo d'occhio.

Quando ciò accade, i pattern attivi di F # possono essere un ottimo modo per dare nomi significativi alla logica di matching, che semplifica il codice e consente anche il riutilizzo.

Examples

Opzioni di corrispondenza

La corrispondenza del modello può essere utile per gestire le opzioni:

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

La corrispondenza del modello verifica che l'intero dominio sia coperto

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

produce un avvertimento

> C: \ Programmi (x86) \ Microsoft VS Code \ Untitled-1 (2,7): avviso FS0025: corrispondenze di pattern incomplete su questa espressione. Ad esempio, il valore 'false' può indicare un caso non coperto dal / i motivo / i.

Questo perché non tutti i possibili valori bool erano coperti.

i bool possono essere elencati esplicitamente ma gli ints sono più difficili da elencare

```
let x = 5
match x with
```

```
| 1 -> printfn "x is 1"
| 2 -> printfn "x is 2"
| _ -> printfn "x is something else"
```

qui usiamo il carattere _ speciale. Il _ corrisponde a tutti gli altri casi possibili.

Il _ può metterti nei guai

considera un tipo che creiamo noi stessi assomiglia a questo

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
```

Potremmo scrivere una corrispondenza con expession che assomiglia a questo

```
match sobriety with
| Sober -> printfn "drive home"
| _ -> printfn "call an uber"
```

Il codice sopra ha un senso. Assumiamo che se non sei sobrio dovresti chiamare un uber, quindi usiamo il _ per denotarlo

In seguito ci rifattiamo il codice su questo

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
    | Unconscious
```

Il compilatore F # dovrebbe darci un avvertimento e chiederci di refactoring la nostra espressione di corrispondenza per avere la persona di cercare cure mediche. Invece l'espressione della corrispondenza tratta silenziosamente la persona incosciente come se fossero solo alticcia. Il punto è che dovresti scegliere di elencare esplicitamente i casi quando possibile per evitare errori logici.

I casi vengono valutati dall'alto verso il basso e viene utilizzata la prima corrispondenza

Utilizzo errato:

Nel seguente frammento, l'ultima partita non verrà mai utilizzata:

```
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"
```

stampe

> x è tutto ciò che non era elencato sopra

Uso corretto:

Qui, sia `x = 1` che `x = 4` colpiranno i loro casi specifici, mentre tutto il resto passerà al caso predefinito `_` :

```
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"
```

stampe

> x è 4

Quando le guardie ti permettono di aggiungere condizionali arbitrari

```
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"
```

Leggi Pattern Matching online: https://riptutorial.com/it/fsharp/topic/1335/pattern-matching

Capitolo 20: Porting C # a F

Examples

pocos

Alcuni dei tipi più semplici di classi sono i POCO.

```
// C#
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }
}
```

In F # 3.0, sono state introdotte proprietà automatiche simili alle proprietà automatiche di C #,

```
// F#
type Person() =
    member val FirstName = "" with get, set
    member val LastName = "" with get, set
    member val BirthDay = System.DateTime.Today with get, set
```

La creazione di un'istanza di entrambi è simile,

```
// C#
var person = new Person { FirstName = "Bob", LastName = "Smith", Birthday = DateTime.Today };
// F#
let person = new Person(FirstName = "Bob", LastName = "Smith")
```

Se è possibile utilizzare valori immutabili, un tipo di record è F # molto più idiomatico.

```
type Person = {
    FirstName:string;
    LastName:string;
    Birthday:System.DateTime
}
```

E questo record può essere creato:

```
let person = { FirstName = "Bob"; LastName = "Smith"; Birthday = System.DateTime.Today }
```

I record possono anche essere creati sulla base di altri record specificando il record esistente e aggiungendo `with` , quindi un elenco di campi da sovrascrivere:

```
let formal = { person with FirstName = "Robert" }
```

Classe Implementazione di un'interfaccia

Le classi implementano un'interfaccia per soddisfare il contratto dell'interfaccia. Ad esempio, una classe C # può implementare `IDisposable` ,

```
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();
        }
    }
}
```

Per implementare un'interfaccia in F #, usa l' `interface` nella definizione del tipo,

```
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 -> ()
```

Leggi Porting C # a F # online: https://riptutorial.com/it/fsharp/topic/6828/porting-c-sharp-a-f-sharp

Capitolo 21: Processore di cassette postali

Osservazioni

`MailboxProcessor` gestisce una coda di messaggi interna, in cui più produttori possono pubblicare messaggi utilizzando varie varianti del metodo `Post` . Questi messaggi vengono quindi recuperati ed elaborati da un singolo utente (a meno che non lo si implementi diversamente) utilizzando le varianti `Retrieve` e `Scan` . Di default sia la produzione che il consumo dei messaggi sono thread-safe.

Di default non esiste una gestione degli errori fornita. Se viene generata un'eccezione non rilevata all'interno del corpo del processore, la funzione body termina, tutti i messaggi nella coda andranno persi, non è più possibile inviare messaggi e il canale di risposta (se disponibile) riceverà un'eccezione anziché una risposta. Devi fornire tutta la gestione degli errori da solo nel caso in cui questo comportamento non si adatti al tuo caso d'uso.

Examples

Basic Hello World

Iniziamo con la creazione di un semplice "Hello world!" `MailboxProcessor` che elabora un tipo di messaggio e stampa i saluti.

Avrai bisogno del tipo di messaggio. Può essere qualsiasi cosa, ma le Unioni Discriminate sono una scelta naturale in quanto elencano tutti i possibili casi in un unico posto e puoi facilmente utilizzare la corrispondenza dei modelli durante l'elaborazione.

```
// In this example there is only a single case, it could technically be just a string
type GreeterMessage = SayHelloTo of string
```

Ora definisci il processore stesso. Questo può essere fatto con `MailboxProcessor<'message>.Start` metodo statico che restituisce un processore avviato pronto a fare il suo lavoro. È anche possibile utilizzare il costruttore, ma in seguito è necessario assicurarsi di avviare il processore in un secondo momento.

```
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 ())
```

Il parametro `Start` è una funzione che prende un riferimento al `MailboxProcessor` stesso (che non esiste ancora come lo si sta creando, ma sarà disponibile una volta eseguita la funzione). Ciò consente di accedere ai vari metodi di `Receive` e `Scan` per accedere ai messaggi dalla casella di posta. All'interno di questa funzione, puoi eseguire qualsiasi elaborazione di cui hai bisogno, ma un approccio normale è un ciclo infinito che legge i messaggi uno ad uno e si richiama dopo ognuno di essi.

Ora il processore è pronto, ma non fa niente! Perché? È necessario inviare un messaggio per elaborare. Questo è fatto con le varianti del metodo `Post` - usiamo il più elementare, fire-and-forget.

```
processor.Post(SayHelloTo "Alice")
```

Questo inserisce un messaggio nella coda interna del `processor` , nella cassetta postale, e ritorna immediatamente in modo che il codice chiamante possa continuare. Una volta che il processore ha recuperato il messaggio, lo elaborerà, ma verrà eseguito in modo asincrono per pubblicarlo e molto probabilmente verrà fatto su un thread separato.

Molto presto dovresti vedere il messaggio `"Hi, Alice! This is mailbox processor's inner loop!"` stampato sull'output e sei pronto per campioni più complicati.

Gestione statale mutabile

I processori delle cassette postali possono essere utilizzati per gestire lo stato mutabile in modo trasparente e sicuro per i thread. Costruiamo un semplice contatore.

```
// 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
```

Ora generiamo alcune operazioni

```
processor.Post(Increment)
processor.Post(Increment)
processor.Post(Decrement)
processor.Post(Increment)
```

E vedrai il seguente registro

```
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
```

Concorrenza

Poiché il processore della casella di posta elabora i messaggi uno per uno e non c'è interleaving, è possibile anche produrre i messaggi da più thread e non si vedranno i problemi tipici delle operazioni perse o duplicate. Non è possibile che un messaggio utilizzi lo stato precedente di altri messaggi, a meno che non si implementi in modo specifico il processore.

```
let processor = createProcessor 0

[ async { processor.Post(Increment) }
  async { processor.Post(Increment) }
  async { processor.Post(Decrement) }
  async { processor.Post(Decrement) } ]
|> Async.Parallel
|> Async.RunSynchronously
```

Tutti i messaggi sono pubblicati da diversi thread. L'ordine in cui i messaggi vengono inviati alla casella di posta non è deterministico, quindi l'ordine di elaborazione non è deterministico, ma poiché il numero complessivo di incrementi e decrementi è bilanciato, lo stato finale è 0, indipendentemente dall'ordine e da quale thread sono stati inviati i messaggi.

Vero stato mutabile

Nell'esempio precedente abbiamo solo simulato lo stato mutabile passando il parametro del ciclo ricorsivo, ma il processore di cassette postali ha tutte queste proprietà anche per uno stato veramente mutabile. Questo è importante quando si mantiene uno stato ampio e l'immutabilità

non è pratica per motivi di prestazioni.

Possiamo riscrivere il nostro contatore alla seguente implementazione

```
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 ())
```

Anche se questo sicuramente non sarebbe thread-safe se lo stato del contatore fosse stato modificato direttamente da più thread, è possibile vedere utilizzando i messaggi di messaggi paralleli della sezione precedente che il processore della casella di posta elabora i messaggi uno dopo l'altro senza interleaving, quindi ogni messaggio usa il valore più attuale.

Valori di ritorno

È possibile restituire in modo asincrono un valore per ogni messaggio elaborato se si invia un `AsyncReplyChannel<'a>` come parte del messaggio.

```
type MessageWithResponse = Message of InputData * AsyncReplyChannel<OutputData>
```

Quindi il processore della casella di posta può utilizzare questo canale durante l'elaborazione del messaggio per inviare un valore al chiamante.

```
let! message = inbox.Receive()
match message with
| MessageWithResponse (data, r) ->
    // ...process the data
    let output = ...
    r.Reply(output)
```

Ora per creare un messaggio, hai bisogno di `AsyncReplyChannel<'a>` - che cos'è e come crei un'istanza di lavoro? Il modo migliore è lasciare che MailboxProcessor lo fornisca per te ed estrai la risposta a un più comune `Async<'a>` . Questo può essere fatto usando, ad esempio, il metodo `PostAndAsynReply` , dove non si pubblica il messaggio completo, ma invece una funzione di tipo (nel nostro caso) `AsyncReplyChannel<OutputData> -> MessageWithResponse` :

```
let! output = processor.PostAndAsyncReply(r -> MessageWithResponse(input, r))
```

Questo invierà il messaggio in coda e attenderà la risposta, che arriverà quando il processore arriva a questo messaggio e risponde utilizzando il canale.

Esiste anche una variante sincrona `PostAndReply` che blocca il thread chiamante finché il processore non risponde.

Elaborazione dei messaggi fuori ordine

È possibile utilizzare i metodi di `Scan` o `TryScan` per cercare messaggi specifici nella coda ed elaborarli indipendentemente dal numero di messaggi che li precedono. Entrambi i metodi esaminano i messaggi nella coda nell'ordine in cui sono arrivati e cercheranno un messaggio specificato (fino al timeout opzionale). Nel caso in cui non vi sia alcun messaggio di questo tipo, `TryScan` restituirà None, mentre `Scan` continuerà ad attendere fino a quando il messaggio non arriva o l'operazione `TryScan` .

Vediamolo in pratica. Vogliamo che il processore elabori `RegularOperations` quando può, ma ogni volta che c'è un `PriorityOperation` , dovrebbe essere elaborato il prima possibile, indipendentemente da quante altre `RegularOperations` sono in coda.

```
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())
```

Leggi Processore di cassette postali online: https://riptutorial.com/it/fsharp/topic/9409/processore-di-cassette-postali

Capitolo 22: Records

Examples

Aggiungi funzioni membro ai record

```
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
```

Utilizzo di base

```
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
```

Leggi Records online: https://riptutorial.com/it/fsharp/topic/1136/records

Capitolo 23: Riflessione

Examples

Riflessione robusta usando le citazioni di F

La riflessione è utile ma fragile. Considera questo:

```
let mi  = typeof<System.String>.GetMethod "StartsWith"
```

I problemi con questo tipo di codice sono:

1. Il codice non funziona perché ci sono diversi overload di `String.StartsWith`
2. Anche se non ci sarebbero sovraccarichi in questo momento, le versioni successive della libreria potrebbero aggiungere un sovraccarico che causa un arresto anomalo del runtime
3. Strumenti di refactoring come i `Rename methods` sono infranti di riflessione.

Ciò significa che si verifica un arresto anomalo del runtime per qualcosa che è noto in fase di compilazione. Sembra inopportuno.

Utilizzando le citazioni di `F#` è possibile evitare tutti i problemi di cui sopra. Definiamo alcune funzioni di supporto:

```
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
```

Usiamo le funzioni in questo modo:

```
printfn "%A" <| getMethodInfo <@ "".StartsWith "" @>
printfn "%A" <| getMethodInfo <@ List.singleton 1 @>
printfn "%A" <| getConstructorInfo <@ System.String [||] @>
```

Questo stampa:

```
Boolean StartsWith(System.String)
Void .ctor(Char[])
Microsoft.FSharp.Collections.FSharpList`1[System.Int32] Singleton[Int32](Int32)
```

`<@ ... @>` significa che invece di eseguire l'espressione all'interno di `F#` genera un albero di espressioni che rappresenta l'espressione. `<@ "".StartsWith "" @>` genera un albero di espressioni simile al seguente: `Call (Some (Value ("")), StartsWith, [Value ("")])` . Questo albero di espressioni corrisponde a ciò che `getMethodInfo` aspetta e restituirà le informazioni sul metodo corretto.

Questo risolve tutti i problemi sopra elencati.

Leggi Riflessione online: https://riptutorial.com/it/fsharp/topic/4124/riflessione

Capitolo 24: Sequenza

Examples

Genera sequenze

Esistono diversi modi per creare una sequenza.

Puoi usare le funzioni dal modulo Seq:

```
// 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
```

Puoi anche usare l'espressione di sequenza:

```
// 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 }
}
```

Introduzione alle sequenze

Una sequenza è una serie di elementi che possono essere elencati. È un alias di System.Collections.Generic.IEnumerable e pigro. Memorizza una serie di elementi dello stesso

tipo (può essere qualsiasi valore o oggetto, anche un'altra sequenza). Le funzioni di Seq.module possono essere utilizzate per operare su di esso.

Ecco un semplice esempio di un'enumerazione di sequenza:

```
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
```

Produzione:

```
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; ...]
```

Applica una funzione a ogni elemento di una sequenza usando Seq.map

Seq.filter

Supponiamo di avere una sequenza di numeri interi e di creare una sequenza che contenga solo gli interi pari. Possiamo ottenere quest'ultimo utilizzando la funzione `filter` del modulo Seq. La funzione `filter` ha la firma del tipo `('a -> bool) -> seq<'a> -> seq<'a>` ; questo indica che accetta una funzione che restituisce vero o falso (talvolta chiamato un predicato) per un dato input di tipo `'a` e una sequenza che comprende valori di tipo `'a` per ottenere una sequenza che comprende valori di tipo `'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
```

Infinite sequenze ripetitive

```
let data = [1; 2; 3; 4; 5;]
let repeating = seq {while true do yield! data}
```

Le sequenze ripetute possono essere create usando un'espressione di calcolo `seq {}`

Leggi Sequenza online: https://riptutorial.com/it/fsharp/topic/2354/sequenza

Capitolo 25: Sindacati discriminati

Examples

Denominazione di elementi di tuple all'interno di sindacati discriminati

Quando si definiscono i sindacati discriminati, è possibile assegnare un nome agli elementi dei tipi di tupla e utilizzare questi nomi durante la corrispondenza dei modelli.

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

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

Inoltre, la denominazione degli elementi di unioni discriminate migliora la leggibilità del codice e l'interoperabilità con C # - i nomi forniti verranno utilizzati per i nomi delle proprietà e i parametri dei costruttori. I nomi generati predefiniti nel codice di interoperabilità sono "Elemento", "Elemento1", "Elemento2" ...

Utilizzo discriminatorio di base dell'Unione

I sindacati discriminati in F # offrono un modo per definire tipi che possono contenere un numero qualsiasi di tipi di dati diversi. La loro funzionalità è simile alle unioni C ++ o alle varianti VB, ma con l'ulteriore vantaggio di essere sicuro.

```
// 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
```

Sindacati in stile Enum

Le informazioni di tipo non devono essere incluse nei casi di unione discriminata. Omettendo le informazioni sul tipo è possibile creare un unione che rappresenta semplicemente un insieme di

scelte, simile a un enum.

```
// 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
```

Conversione da e verso le stringhe con Reflection

A volte è necessario convertire un'unione discriminata in e da una stringa:

```
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
```

Unione discriminata caso singolo

Un'unione discriminata in un singolo caso è come qualsiasi altra unione discriminata, tranne che ha un solo caso.

```
// 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
```

È utile per rafforzare la sicurezza del tipo e comunemente usato in F # rispetto a C # e Java, dove la creazione di nuovi tipi comporta un ulteriore sovraccarico.

Le seguenti due definizioni di tipi alternativi provocano la stessa unione discriminata a caso singolo:

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

Utilizzo di unioni discriminate a caso singolo come record

A volte è utile creare tipi di unione con un solo caso per implementare tipi di record:

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)

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

Questi diventano molto utili perché possono essere scomposti tramite la corrispondenza dei pattern nello stesso modo in cui gli argomenti di tuple possono:

```
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

Con l'attributo `RequireQualifiedAccess` , i casi di unione devono essere indicati come `MyUnion.MyCase` anziché solo `MyCase` . Ciò impedisce le collisioni di nomi nello spazio dei nomi o nel modulo allegato:

```
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
```

Se, ad esempio, `System` è stato aperto, `Single` fa riferimento a `System.Single` . Non vi è alcuna collisione con il caso del sindacato `Requirements.Single` .

Sindacati discriminati ricorsivi

Tipo ricorsivo

I sindacati discriminati possono essere ricorsivi, cioè possono riferirsi a se stessi nella loro definizione. L'esempio principale qui è un albero:

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

Ad esempio, definiamo il seguente albero:

```
    1
  2   5
3   4
```

Possiamo definire questo albero usando la nostra unione discriminata ricorsiva come segue:

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

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

L'iterazione sull'albero è quindi solo una questione di corrispondenza del modello:

```
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]
```

Tipi ricorsivi mutuamente dipendenti

Un modo per ottenere la ricorsione consiste nell'annidare tipi mutualmente dipendenti.

```
// 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
```

La definizione di un tipo di record direttamente all'interno di un'unione discriminata è deprecata:

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

È possibile utilizzare la parola chiave `and` per catene le definizioni reciprocamente dipendenti:

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

Leggi Sindacati discriminati online: https://riptutorial.com/it/fsharp/topic/1025/sindacati-discriminati

Capitolo 26: stringhe

Examples

Stringhe letterali

```
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
```

Semplice formattazione di stringhe

Di conseguenza, esistono diversi modi per formattare e ottenere una stringa.

Il modo .NET è utilizzando `String.Format` o `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 # ha anche funzioni per formattare una stringa in stile C. Esistono equivalenti per ciascuna funzione .NET:

- `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
```

L'utilizzo di tali funzioni al posto delle funzioni .NET offre alcuni vantaggi:

- Digitare sicurezza
- Applicazione parziale
- Supporto di tipo nativo F #

Leggi stringhe online: https://riptutorial.com/it/fsharp/topic/1397/stringhe

Capitolo 27: Suggerimenti e trucchi per le prestazioni F

Examples

Usando la ricorsione in coda per un'iterazione efficiente

Provenienti da linguaggi imperativi, molti sviluppatori si chiedono come scrivere un `for-loop` che esce presto poiché `F#` non supporta `break` , `continue` o `return` . La risposta in `F#` consiste nell'usare la ricorsione in coda che è un modo flessibile e idiomatico per iterare pur fornendo prestazioni eccellenti.

Diciamo che vogliamo implementare `tryFind` per `List` . Se `F#` supportato `return` avremmo scritto `tryFind` un po 'come questo:

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

Questo non funziona in `F#` . Invece scriviamo la funzione usando la ricorsione in coda:

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

La ricorsione in coda è performante in `F#` perché quando il compilatore `F#` rileva che una funzione è ricorsiva in coda, riscrive la ricorsione in un efficiente `while-loop` . Usando `ILSpy` possiamo vedere che questo è vero per il nostro `loop` funzioni:

```
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;
}
```

A parte alcuni compiti non necessari (che si spera che il JIT-er elimini) questo è essenzialmente un ciclo efficiente.

Inoltre, la ricorsione in coda è idiomatica per `F#` quanto ci consente di evitare lo stato mutabile. Considera una funzione `sum` che somma tutti gli elementi in una `List` . Un primo tentativo ovvio sarebbe questo:

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

Se riscriviamo il loop in coda-ricorsione possiamo evitare lo stato mutabile:

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

Per efficienza, il compilatore `F#` trasforma in un `while-loop` che utilizza lo stato mutabile.

Misura e verifica le ipotesi di rendimento

Questo esempio è scritto con `F#` in mente ma le idee sono applicabili in tutti gli ambienti

La prima regola quando si ottimizzano le prestazioni consiste nel non fare affidamento sull'ipotesi; Misurare sempre e verificare le ipotesi.

Poiché non stiamo scrivendo direttamente il codice macchina, è difficile prevedere come il compilatore e JIT: er trasformino il programma in codice macchina. Ecco perché dobbiamo misurare il tempo di esecuzione per vedere che otteniamo il miglioramento delle prestazioni che ci aspettiamo e verificare che il programma attuale non contenga alcun overhead nascosto.

La verifica è il processo abbastanza semplice che coinvolge il reverse engineering dell'eseguibile usando strumenti come ILSpy . Il JIT: er complica la verifica in quanto vedere il codice macchina effettivo è difficile ma fattibile. Tuttavia, di solito l'esame del `IL-code` fornisce grandi vantaggi.

Il problema più difficile è Misurare; più difficile perché è difficile configurare situazioni realistiche che consentano di misurare i miglioramenti nel codice. La misurazione continua è inestimabile.

Analizzando semplici funzioni F #

Esaminiamo alcune semplici funzioni `F#` che accumulano tutti gli interi in `1..n` scritti in vari modi. Poiché l'intervallo è una semplice serie aritmetica, il risultato può essere calcolato direttamente,

ma per lo scopo di questo esempio, iteriamo sull'intervallo.

Per prima cosa definiamo alcune utili funzioni per misurare il tempo impiegato da una funzione:

```
// 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` esegue ripetutamente un'azione, è necessario eseguire i test per alcune centinaia di millisecondi per ridurre la varianza.

Quindi definiamo alcune funzioni che accumulano tutti gli interi in `1..n` in modi diversi.

```
// 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
```

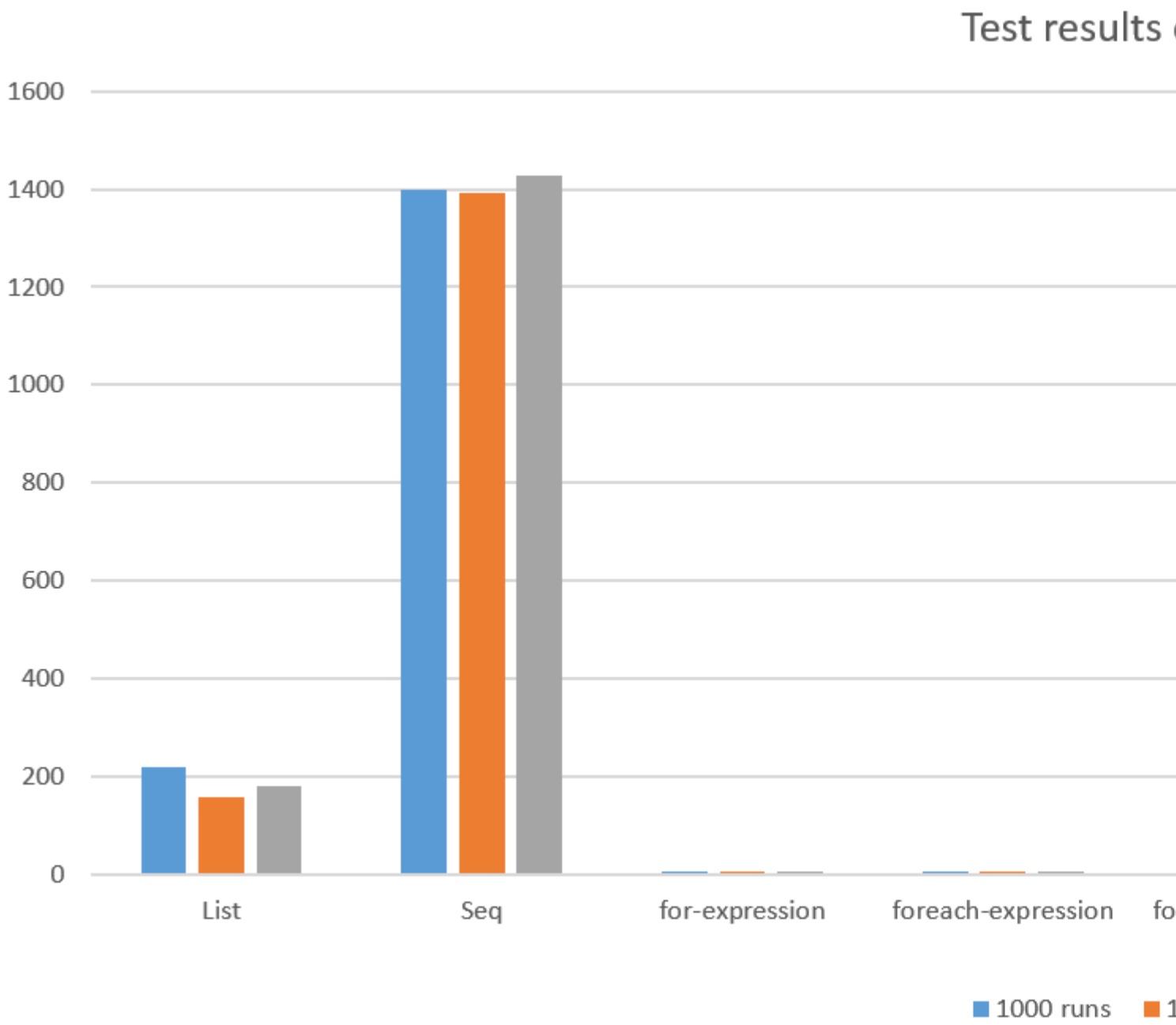
```
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum

// 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
```

Assumiamo che il risultato sia lo stesso (ad eccezione di una delle funzioni che utilizza l'incremento di `2`) ma c'è una differenza nelle prestazioni. Per misurare questo è definita la seguente funzione:

```
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
```

Il risultato del test durante l'esecuzione su .NET 4.5.2 x64:



Vediamo una differenza drammatica e alcuni risultati sono inaspettatamente cattivi.

Diamo un'occhiata ai casi brutti:

Elenco

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

Quello che succede qui è una lista completa contenente tutti gli interi `1..n` viene creata e ridotta usando una somma. Questo dovrebbe essere più costoso della semplice iterazione e accumulo nell'intervallo, sembra circa 42 volte più lento del ciclo for.

Inoltre, possiamo vedere che il GC ha eseguito circa 100x durante l'esecuzione del test perché il

codice ha allocato molti oggetti. Anche questo costa CPU.

Seq

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

La versione `Seq` non assegna un `List` completo quindi è un po 'sorprendente che questo ~ 270x più lento del ciclo for. Inoltre, vediamo che il GC ha eseguito 661x.

`Seq` è inefficiente quando la quantità di lavoro per articolo è molto piccola (in questo caso aggregando due numeri interi).

Il punto è non usare mai `Seq` . Il punto è misurare.

(**modifica manofstick:** `Seq.init` è il colpevole di questo grave problema di prestazioni. È molto più efficace usare l'espressione `{ 0 .. n }` invece di `Seq.init (n+1) id` . Questo diventerà molto più efficiente ancora quando questo PR viene unito e rilasciato Anche dopo il rilascio, il `Seq.init ... |> Seq.sum` originale `Seq.init ... |> Seq.sum` sarà ancora lento, ma in qualche modo contro-intuitivamente, `Seq.init ... |> Seq.map id |> Seq.sum` sarà abbastanza veloce, per mantenere la compatibilità con l'implementazione di `Seq.init` , che non calcola inizialmente `Current` , ma piuttosto li avvolge in un oggetto `Lazy` - anche se anche questo dovrebbe funzionare un po 'meglio a causa di questo PR Note per l'editore: scusa si tratta di una specie di note vaganti, ma non voglio che le persone vengano messe fuori Seq quando il miglioramento è dietro l'angolo ... *Quando arriveranno i tempi sarebbe bello aggiornare i grafici che sono in questa pagina.*)

foreach-expression su 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
```

La differenza tra queste due funzioni è molto sottile, ma la differenza di prestazioni non è di circa 76x. Perché? Eseguiamo il reverse engineering del codice errato:

```
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` è implementato come un ciclo `while` efficiente ma `for i in [1..n]` viene convertito in:

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

Ciò significa innanzitutto creare un `Seq` su `1..n` e infine chiamare `ToList` .

Costoso.

incremento di foreach-expression di 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
```

Ancora una volta la differenza tra queste due funzioni è sottile ma la differenza di prestazioni è brutale: ~ 25x

Ancora una volta `ILSpy` :

```
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;
}
```

Un `Seq` viene creato su `1..2..n` e poi `1..2..n` su `Seq` usando l'enumeratore.

Ci aspettavamo che `F#` creasse qualcosa del genere:

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

Tuttavia, il compilatore `F#` supporta solo cicli efficienti su intervalli int32 che aumentano di uno. Per tutti gli altri casi ricade su `Operators.OperatorIntrinsics.RangeInt32` . Che spiegherà il prossimo risultato sorprendente

foreach-expression over 64 bit

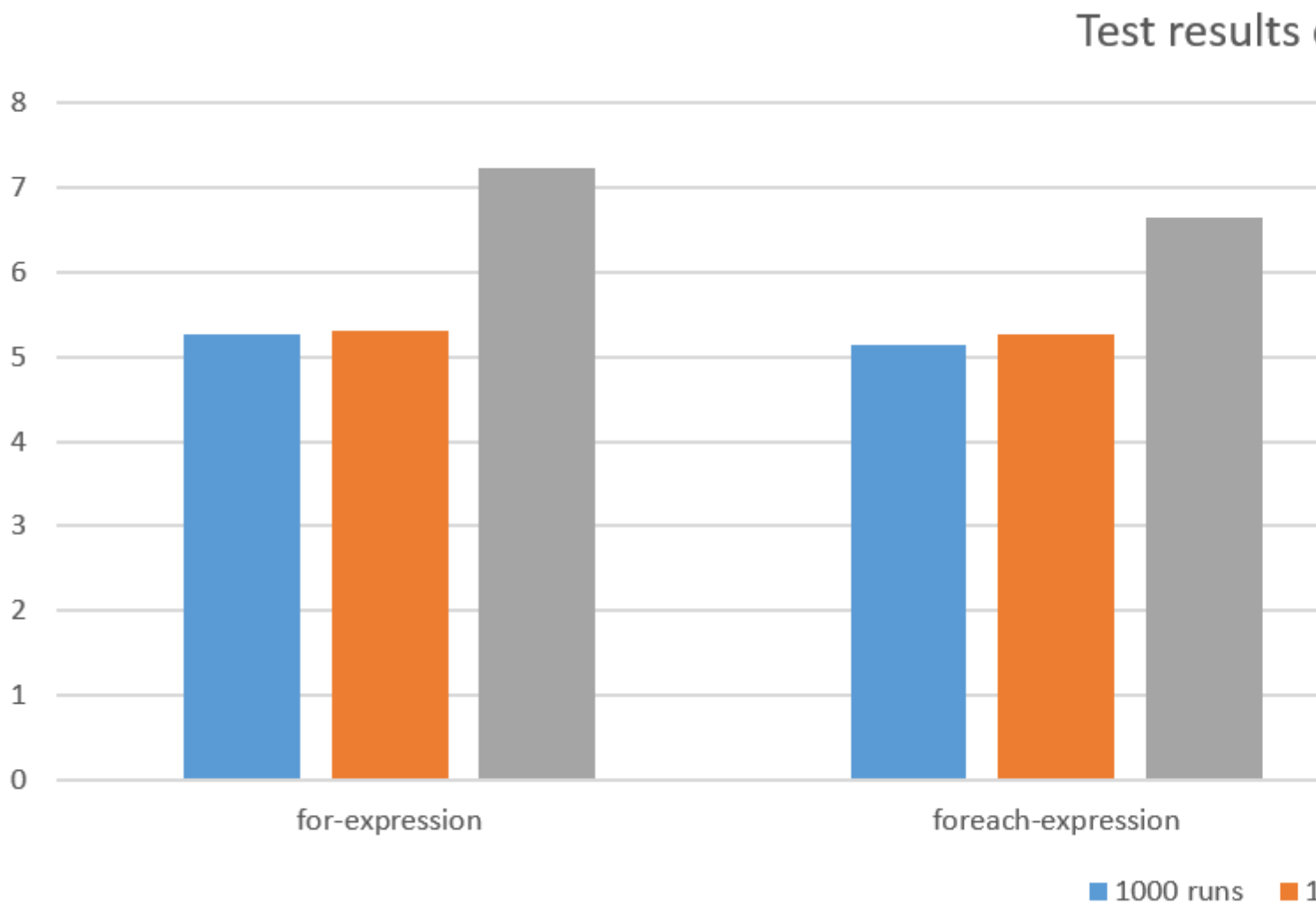
```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int
```

Questo esegue ~ 47 volte più lento del ciclo for, l'unica differenza è che iteriamo su interi a 64 bit. `ILSpy` ci mostra perché:

```
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#` supporta solo cicli efficienti per i numeri `int32` che deve utilizzare il fallback `Operators.OperatorIntrinsics.RangeInt64` .

Gli altri casi si presentano approssimativamente simili:



Il motivo per cui le prestazioni si riducono per le esecuzioni di test più grandi è che il sovraccarico di invocare l' `action` è in aumento man mano che facciamo sempre meno lavoro in `action` .

Il loop verso `0` può talvolta dare benefici in termini di prestazioni poiché potrebbe salvare un registro della CPU, ma in questo caso la CPU ha i registri da risparmiare, quindi non sembra fare la differenza.

Conclusione

La misurazione è importante perché altrimenti potremmo pensare che tutte queste alternative siano equivalenti, ma alcune alternative sono ~ 270 volte più lente di altre.

La fase di verifica implica il reverse engineering, l'eseguibile ci aiuta a spiegare *perché* abbiamo ottenuto o meno le prestazioni che ci aspettavamo. Inoltre, la verifica può aiutarci a prevedere le prestazioni nei casi in cui è troppo difficile eseguire una misurazione adeguata.

È difficile prevedere le prestazioni là sempre Misura, verifica sempre le ipotesi di rendimento.

Confronto tra diverse pipeline di dati F

In `F#` ci sono molte opzioni per la creazione di pipeline di dati, ad esempio: `List` , `Seq` e `Array` .

Quale pipeline di dati è preferibile dall'uso della memoria e dal punto di vista delle prestazioni?

Per rispondere a questo, confronteremo le prestazioni e l'utilizzo della memoria utilizzando diverse pipeline.

Pipeline di dati

Per misurare il sovraccarico, utilizzeremo una pipeline di dati con un costo di cpu basso per articoli elaborati:

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

Creeremo condutture equivalenti per tutte le alternative e le confronteranno.

Variamo la dimensione di `n` ma lasciamo che il numero totale di lavori sia lo stesso.

Alternative alla pipeline di dati

Confronteremo le seguenti alternative:

1. Codice imperativo
2. Matrice (non pigra)
3. Elenco (non pigro)
4. LINQ (Lazy pull stream)
5. Seq (Lazy pull stream)
6. Nessos (Lazy pull / push stream)
7. PullStream (flusso di pull semplicistico)
8. PushStream (flusso push semplicistico)

Sebbene non si tratti di una pipeline di dati, confronteremo il codice `Imperative` dal momento che più strettamente corrisponde al modo in cui la CPU esegue il codice. Questo dovrebbe essere il modo più veloce per calcolare il risultato, consentendoci di misurare il sovraccarico delle prestazioni delle pipeline di dati.

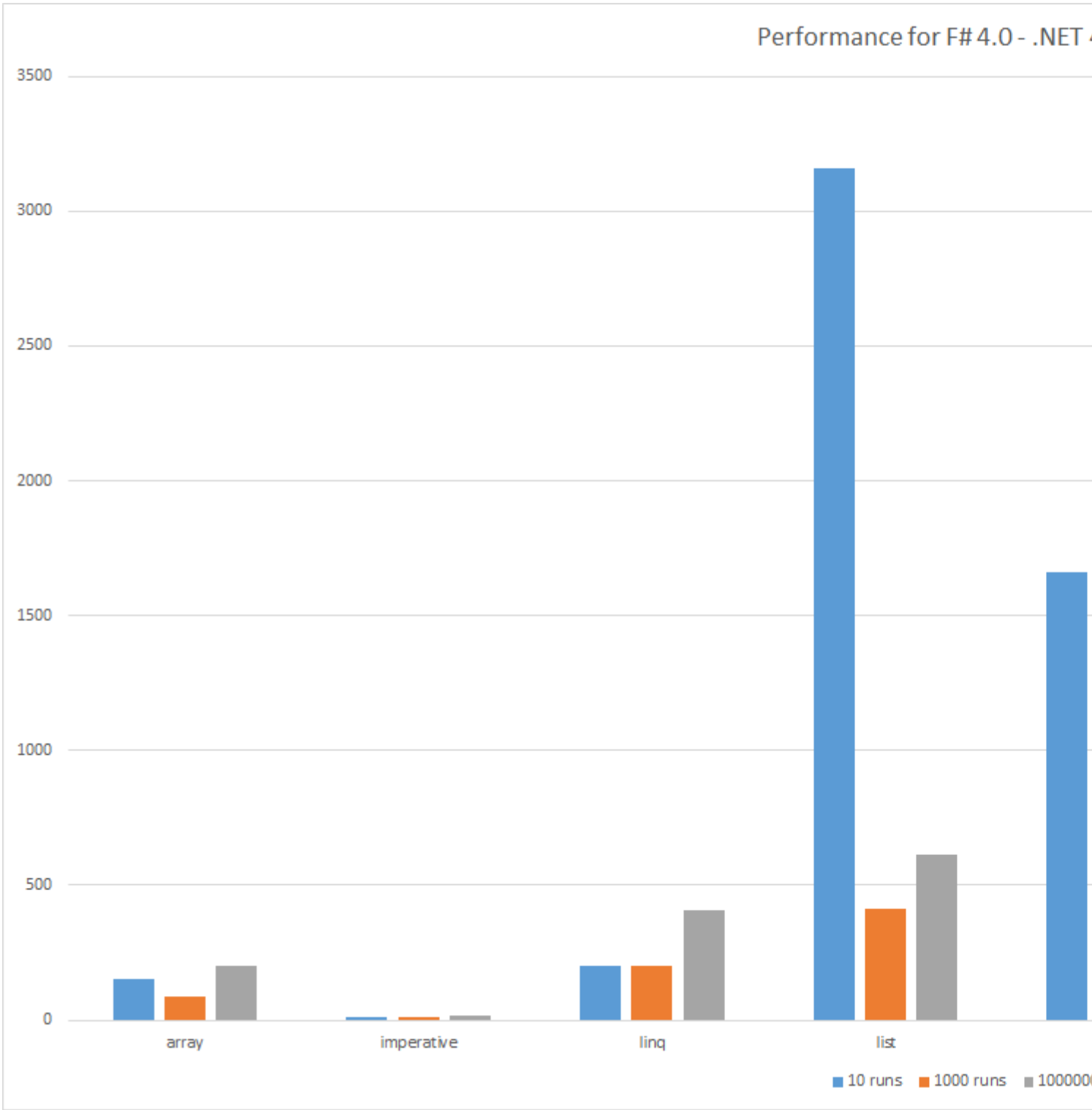
`Array` e `List` calcolano una `Array` / `List` in ogni passaggio, quindi ci aspettiamo un sovraccarico della memoria.

`LINQ` e `Seq` sono entrambi basati su `IEnumerable<'T>` che è lazy pull stream (pull significa che lo stream consumer sta estraendo i dati dallo stream del produttore). Pertanto, ci aspettiamo che le prestazioni e l'utilizzo della memoria siano identici.

`Nessos` è una libreria di stream ad alte prestazioni che supporta sia push & pull (come Java `Stream`).

PullStream e PushStream sono implementazioni semplicistiche dei flussi `Pull` & `Push` .

Prestazioni Risultati dall'esecuzione su: F # 4.0 - .NET 4.6.1 - x64

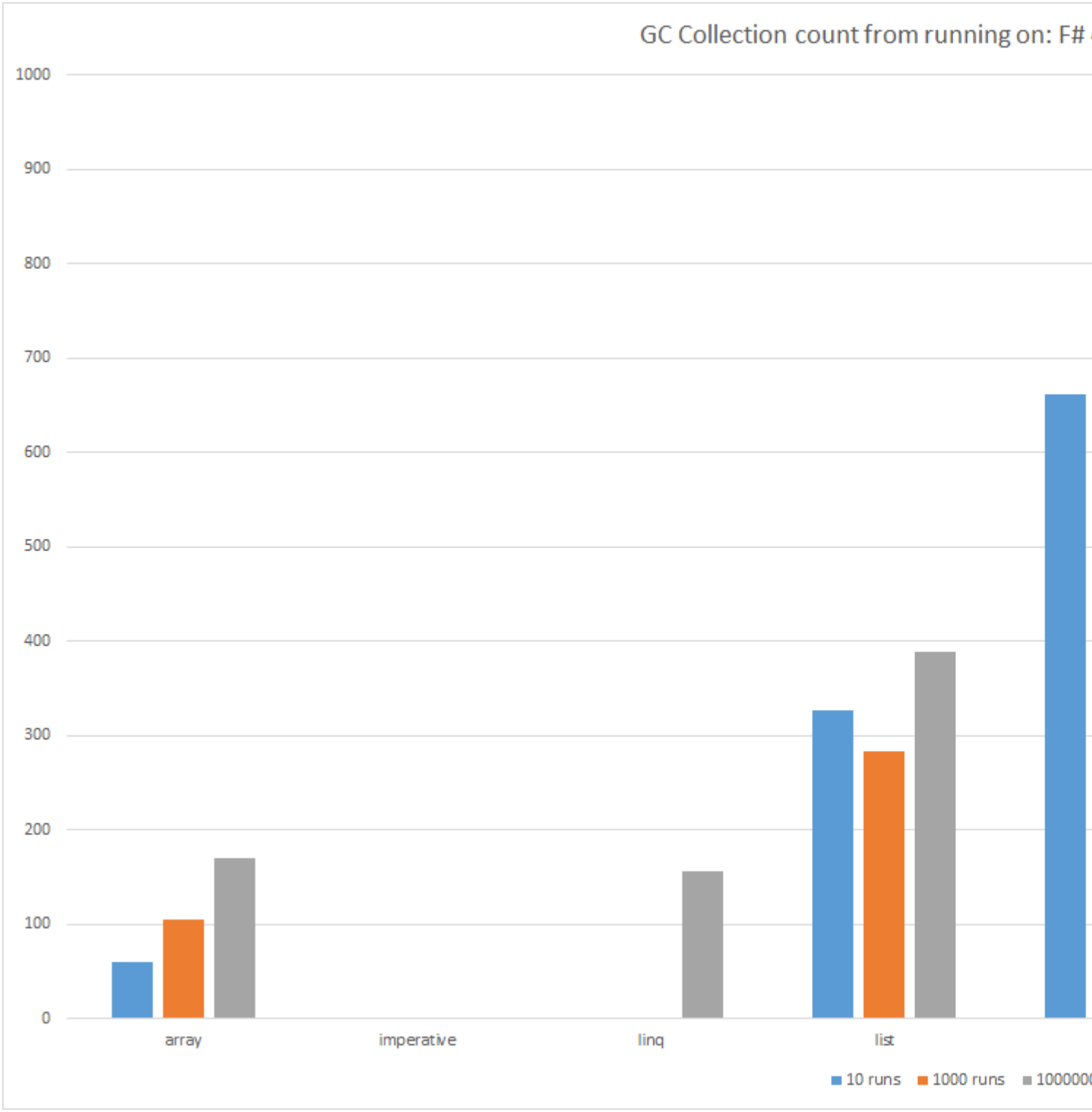


Le barre mostrano il tempo trascorso, più in basso è meglio. La quantità totale di lavoro utile è la stessa per tutti i test, quindi i risultati sono comparabili. Ciò significa anche che poche esecuzioni implicano set di dati più grandi.

Come al solito quando si misura uno vedi risultati interessanti.

1. `List` prestazioni di `List` scadenti vengono confrontate con altre alternative per set di dati di grandi dimensioni. Questo può essere dovuto a `GC` o alla scarsa localizzazione della cache.
2. `Array` prestazioni della `Array` sono migliori del previsto.
3. `LINQ` prestazioni migliori di `Seq` , questo è inaspettato perché entrambi sono basati su `IEnumerable<'T>` . Tuttavia, `Seq` internamente si basa su una generica impementazione per tutti gli algoritmi mentre `LINQ` utilizza algoritmi specializzati.
4. `Push` esegue meglio di `Pull` . Questo è previsto poiché la pipeline di dati push ha meno controlli
5. Le semplicistiche pipeline di dati `Push` sono comparabili a quelle di `Nessos` . Tuttavia, `Nessos` supporta pull e parallelismo.
6. Per le pipeline di dati di piccole dimensioni, le prestazioni di `Nessos` riducono a causa del sovraccarico dell'installazione delle pipeline.
7. Come previsto, il codice `Imperative` ottenuto il meglio

Conteggio delle raccolte GC dall'esecuzione su: F # 4.0 - .NET 4.6.1 - x64



Le barre mostrano il numero totale di conteggi di raccolta `GC` durante il test, più basso è migliore. Questa è una misura del numero di oggetti creati dalla pipeline di dati.

Come al solito quando si misura uno vedi risultati interessanti.

1. È previsto che la `List` crei più oggetti rispetto `Array` perché un `List` è essenzialmente un singolo elenco di nodi collegati. Una matrice è un'area di memoria continua.
2. Guardando i numeri sottostanti sia `List` `Array` costringono 2 collezioni di generazione. Questo tipo di collezione è costoso.
3. `Seq` sta innescando una quantità sorprendente di collezioni. È sorprendentemente persino

peggiore di `List` in questo senso.
4. `LINQ` , `Nessos` , `Push` e `Pull` non attivano raccolte per poche esecuzioni. Tuttavia, gli oggetti sono assegnati in modo che il `GC` alla fine debba essere eseguito.
5. Come previsto dal momento che il codice `Imperative` non ha assegnato alcun oggetto, non sono state attivate raccolte `GC` .

Conclusione

Tutte le pipeline di dati svolgono la stessa quantità di lavoro utile in tutti i casi di test, ma vediamo differenze significative nelle prestazioni e nell'utilizzo della memoria tra le diverse pipeline.

Inoltre, notiamo che il sovraccarico delle pipeline di dati varia a seconda delle dimensioni dei dati elaborati. Ad esempio, per le piccole dimensioni, l' `Array` si comporta abbastanza bene.

Si dovrebbe tenere a mente che la quantità di lavoro svolto in ogni fase della pipeline è molto piccola per misurare il sovraccarico. In situazioni "reali" il sovraccarico di `Seq` potrebbe non avere importanza perché il lavoro effettivo richiede più tempo.

Di maggiore preoccupazione sono le differenze di utilizzo della memoria. `GC` non è gratuito ed è vantaggioso per le applicazioni di lunga durata che riducono la pressione di `GC` .

Per gli sviluppatori di `F#` preoccupati per le prestazioni e l'utilizzo della memoria, si consiglia di controllare Nessos Streams .

Se avete bisogno di prestazioni di alto livello, il codice `Imperative` posizionato strategicamente vale la pena di essere preso in considerazione.

Infine, quando si tratta di prestazioni, non fare supposizioni. Misura e verifica.

Codice sorgente completo:

```
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
```

Leggi Suggerimenti e trucchi per le prestazioni F # online:
https://riptutorial.com/it/fsharp/topic/3562/suggerimenti-e-trucchi-per-le-prestazioni-f-sharp

Capitolo 28: tipi

Examples

Introduzione ai tipi

I tipi possono rappresentare vari tipi di cose. Può essere un singolo dato, un insieme di dati o una funzione.

In F #, possiamo raggruppare i tipi in due categorie .:

- Tipi F #:

```
// 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
```

- Tipi .NET
    - Tipo integrato (int, bool, stringa, ...)
    - Classi, strutture e interfacce
    - I delegati
    - Array

Abbreviazioni di tipo

Le abbreviazioni di tipo ti consentono di creare alias su tipi esistenti per dare loro un significato più significativo.

```
// Name is an alias for a string
type Name = string
```

```
// PhoneNumber is an alias for a string
type PhoneNumber = string
```

Quindi puoi usare l'alias come qualsiasi altro tipo:

```
// 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";}
```

Fai attenzione, gli alias non controllano la coerenza del tipo. Ciò significa che due alias che hanno come target lo stesso tipo possono essere assegnati l'uno all'altro:

```
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";}
```

I tipi sono creati in F # usando la parola chiave type

`F#` usa la parola chiave `type` per creare diversi tipi di tipi.

1. Digita gli alias
2. Tipi sindacali discriminati
3. Registra i tipi
4. Tipi di interfaccia
5. Tipi di classe
6. Tipi di Struct

Esempi con codice `C#` equivalente `C#` ove possibile:

```
// 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
```

Tipo di inferenza

Riconoscimento

Questo esempio è adattato da questo articolo sull'inferenza di tipo

Cos'è il tipo Inferenza?

Type Inference è il meccanismo che consente al compilatore di dedurre quali tipi sono usati e dove. Questo meccanismo è basato su un algoritmo spesso chiamato "Hindley-Milner" o "HM". Vedi sotto alcune delle regole per determinare i tipi di valori semplici e funzionali:

- Guarda i letterali
- Guarda le funzioni e gli altri valori con cui qualcosa interagisce
- Guarda qualsiasi vincolo di tipo esplicito
- Se non ci sono vincoli da nessuna parte, generalizza automaticamente a tipi generici

Guarda i letterali

Il compilatore può dedurre i tipi osservando i letterali. Se il letterale è un int e si aggiunge "x" ad esso, quindi "x" deve essere anche un int. Ma se il letterale è un float e stai aggiungendo "x" ad esso, anche "x" deve essere un float.

Ecco alcuni esempi:

```
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"
```

Guarda le funzioni e gli altri valori con cui interagisce

Se non ci sono letterali da nessuna parte, il compilatore prova ad elaborare i tipi analizzando le funzioni e gli altri valori con cui interagiscono.

```
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
```

Guarda qualsiasi vincolo o annotazione di tipo esplicito

Se sono specificati vincoli di tipo esplicito o annotazioni, il compilatore li utilizzerà.

```
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
```

Generalizzazione automatica

Se dopo tutto questo, non sono stati trovati vincoli, il compilatore rende semplicemente i tipi generici.

```
let inferGeneric x = x
let inferIndirectGeneric x = inferGeneric x
let inferIndirectGenericAgain x = (inferIndirectGeneric x).ToString()
```

Cose che possono andare storte con l'inferenza di tipo

L'inferenza di tipo non è perfetta, ahimè. A volte il compilatore non ha la minima idea di cosa fare. Di nuovo, capire cosa sta succedendo ti aiuterà davvero a stare calmo invece di voler uccidere il compilatore. Ecco alcuni dei principali motivi per gli errori di tipo:

- Dichiarazioni fuori servizio
- Non abbastanza informazioni
- Metodi sovraccaricati

Dichiarazioni fuori servizio

Una regola di base è che devi dichiarare le funzioni prima che vengano utilizzate.

Questo codice ha esito negativo:

```
let square2 x = square x   // fails: square not defined
let square x = x * x
```

Ma questo è ok:

```
let square x = x * x
let square2 x = square x   // square already defined earlier
```

Dichiarazioni ricorsive o simultanee

Una variante del problema "fuori servizio" si verifica con funzioni o definizioni ricorsive che devono fare riferimento l'una all'altra. In questo caso nessuna quantità di riordino sarà d'aiuto: dobbiamo usare parole chiave aggiuntive per aiutare il compilatore.

Quando una funzione viene compilata, l'identificatore della funzione non è disponibile per il corpo. Quindi, se si definisce una funzione ricorsiva semplice, si otterrà un errore del compilatore. La correzione consiste nell'aggiungere la parola chiave "rec" come parte della definizione della funzione. Per esempio:

```
// 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
```

Ecco la versione fissa con "rec fib" aggiunta per indicare che è ricorsiva:

```
let rec fib n =              // LET REC rather than LET
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

Non abbastanza informazioni

A volte, il compilatore non ha abbastanza informazioni per determinare un tipo. Nell'esempio seguente, il compilatore non conosce il tipo su cui il metodo Length deve funzionare. Ma non può nemmeno renderlo generico, quindi si lamenta.

```
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 ...
```

Questi tipi di errore possono essere corretti con annotazioni esplicite.

```
let stringLength (s:string) = s.Length
```

Metodi sovraccaricati

Quando si chiama una classe o un metodo esterno in .NET, si verificano spesso errori dovuti a sovraccarico.

In molti casi, come nell'esempio di concat qui sotto, dovresti annotare esplicitamente i parametri della funzione esterna in modo che il compilatore sappia quale metodo di overload debba chiamare.

```
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
```

A volte i metodi in overload hanno nomi di argomenti diversi, nel qual caso puoi anche dare al compilatore un indizio nominando gli argomenti. Ecco un esempio per il costruttore StreamReader.

```
let makeStreamReader x = new System.IO.StreamReader(x)        //fails
let makeStreamReader x = new System.IO.StreamReader(path=x)   //works
```

Leggi tipi online: https://riptutorial.com/it/fsharp/topic/3559/tipi

Capitolo 29: Tipi di opzioni

Examples

Definizione dell'opzione

`Option` è un'unione discriminata con due casi, `None` o `Some` .

```
type Option<'T> = Some of 'T | None
```

Usa l'opzione <'T> su valori nulli

Nei linguaggi di programmazione funzionale come `F#` `null` valori `null` sono considerati potenzialmente dannosi e di stile scadente (non idiomatici).

Considera questo codice `C#` :

```
string x = SomeFunction ();
int    l = x.Length;
```

`x.Length` getterà se `x` è `null` aggiungiamo protezione:

```
string x = SomeFunction ();
int    l = x != null ? x.Length : 0;
```

O:

```
string x = SomeFunction () ?? "";
int    l = x.Length;
```

O:

```
string x = SomeFunction ();
int    l = x?.Length;
```

In `F#` idiomatic `null` valori `null` non sono usati, quindi il nostro codice assomiglia a questo:

```
let x = SomeFunction ()
let l = x.Length
```

Tuttavia, a volte è necessario rappresentare valori vuoti o non validi. Quindi possiamo usare `Option<'T>` :

```
let SomeFunction () : string option = ...
```

`SomeFunction` restituisce `Some` valore di `string` o `None` . Estraiamo il valore della `string` utilizzando la

corrispondenza del modello

```
let v =
  match SomeFunction () with
  | Some x  -> x.Length
  | None    -> 0
```

Il motivo per cui questo codice è meno fragile di:

```
string x = SomeFunction ();
int    l = x.Length;
```

È perché non possiamo chiamare la `Length` su `string option` . Abbiamo bisogno di estrarre il valore della `string` usando la corrispondenza del modello e così facendo ci garantisce che il valore della `string` è sicuro da usare.

Il modulo opzionale consente la programmazione orientata alle ferrovie

La gestione degli errori è importante ma può trasformare un elegante algoritmo in un pasticcio. `ROP` (Railway Oriented Programming) è utilizzato per rendere la gestione degli errori elegante e componibile.

Considera la semplice funzione `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
```

Lo scopo di `f` è quello di analizzare il valore della `string` input (se c'è `Some`) in un `int` . Se `int` è maggiore di `0` lo gettiamo in un `float` . In tutti gli altri casi eseguiamo il salvataggio con `None` .

Sebbene, una funzione estremamente semplice, la `match` annidata diminuisce significativamente la leggibilità.

`ROP` osserva che abbiamo due tipi di percorsi di esecuzione nel nostro programma

1. Percorso felice: alla fine calcolerà `Some` valore
2. Percorso errore: tutti gli altri percorsi generano `None`

Poiché i percorsi di errore sono più frequenti tendono a prendere il controllo del codice. Vorremmo che il codice percorso felice fosse il percorso di codice più visibile.

Una funzione equivalente `g` usa `ROP` potrebbe assomigliare a questa:

```
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
```

Assomiglia molto a come tendiamo a elaborare elenchi e sequenze in `F#` .

Si può vedere `Option<'T>` come una `List<'T>` che può contenere solo `0` o `1` elemento dove `Option.bind` si comporta come `List.pick` (concettualmente `Option.bind` meglio a `List.collect` ma `List.pick` potrebbe essere più facile da capire).

`bind` , `filter` e `map` gestisce i percorsi di errore e `g` contengono solo il codice percorso felice.

Tutte le funzioni che accettano direttamente l' `Option<_>` e restituiscono l' `Option<_>` sono direttamente componibili con `|>` e `>>` .

`ROP` quindi aumenta la leggibilità e la componibilità.

Utilizzo dei tipi di opzioni da C

Non è una buona idea esporre i tipi di opzioni al codice C #, poiché C # non ha un modo per gestirli. Le opzioni sono o per introdurre `FSharp.Core` come dipendenza nel tuo progetto C # (che è quello che dovresti fare se stai consumando una libreria F # *non* progettata per l'interoperabilità con C #), o per modificare i valori `None` su `null` .

Pre-F # 4.0

Il modo per farlo è creare una funzione di conversione personalizzata:

```
let OptionToObject opt =
    match opt with
    | Some x -> x
    | None -> null
```

Per i tipi di valore dovresti ricorrere al pugilato o all'utilizzo di `System.Nullable` .

```
let OptionToNullable x =
    match x with
    | Some i -> System.Nullable i
    | None -> System.Nullable ()
```

F # 4.0

In F # 4.0, le funzioni `ofObj` , `toObj` , `ofNullable` e `toNullable` introdotte nel modulo `Option` . In F # interattiva possono essere usati come segue:

```
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
```

Si noti che `None` compila in `null` internamente. Tuttavia, per quanto riguarda F # è un `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
```

Leggi Tipi di opzioni online: https://riptutorial.com/it/fsharp/topic/3175/tipi-di-opzioni

Capitolo 30: Tipo Provider

Examples

Utilizzo del provider di tipi CSV

Dato il seguente file CSV:

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

Puoi leggere i dati con il seguente script:

```
#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
```

Utilizzando il provider di tipo WMI

Il provider del tipo WMI consente di interrogare i servizi WMI con una forte digitazione.

Per generare i risultati di una query WMI come JSON,

```
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)
```

Leggi Tipo Provider online: https://riptutorial.com/it/fsharp/topic/1631/tipo-provider

Capitolo 31: Unità di misura

Osservazioni

Unità in fase di esecuzione

Le unità di misura vengono utilizzate solo per il controllo statico da parte del compilatore e non sono disponibili in fase di esecuzione. Non possono essere utilizzati in riflessione o in metodi come `ToString` .

Ad esempio, C # fornisce un `double` senza unità per un campo di tipo `float<m>` definito da ed esposto da una libreria F #.

Examples

Garantire le unità coerenti nei calcoli

Le unità di misura sono ulteriori annotazioni di tipo che possono essere aggiunte a float o interi. Possono essere utilizzati per verificare in fase di compilazione che i calcoli utilizzano le unità in modo coerente.

Per definire annotazioni:

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

Una volta definite, è possibile utilizzare annotazioni per verificare che un'espressione produca il tipo previsto.

```
// 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
```

Conversioni tra unità

```
[<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>
```

Nota che il compilatore F # non sa che `1<m>` uguale a `100<cm>` . Per quanto a lui importa, le unità sono tipi separati. Puoi scrivere funzioni simili per convertire da metri a chilogrammi, e il compilatore non si preoccuperebbe.

```
[<Measure>] type kg

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

Non è possibile definire unità di misura come multipli di altre unità come

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

Tuttavia, definire le unità "per qualcosa", per esempio Hertz, misurare la frequenza, è semplicemente "al secondo", è piuttosto semplice.

```
// 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
```

Utilizzare LanguagePrimitives per conservare o impostare le unità

Quando una funzione non conserva automaticamente le unità a causa delle operazioni di livello inferiore, il modulo `LanguagePrimitives` può essere utilizzato per impostare le unità sulle primitive che le supportano:

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

```
    LanguagePrimitives.Float32WithMeasure (float32 x)
```

Per assegnare unità di misura a un valore in virgola mobile a precisione doppia, basta moltiplicare per uno con le unità corrette:

```
[<Measure>]
type USD

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

Per assegnare unità di misura ad un valore senza unità che non è System.Double, ad esempio, arrivando da una libreria scritta in un'altra lingua, utilizzare una conversione:

```
open LanguagePrimitives

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

Ecco i tipi di funzione riportati da F # interattivo:

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

Parametri di tipo unità di misura

L'attributo `[<Measure>]` può essere utilizzato sui parametri del tipo per dichiarare tipi generici rispetto alle unità di misura:

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

Utilizzo del test:

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

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

Utilizzare tipi di unità standardizzati per mantenere la compatibilità

Ad esempio, i tipi per unità SI sono stati standardizzati nella libreria di base F #, in `Microsoft.FSharp.Data.UnitSystems.SI` . Aprire lo spazio dei nomi secondario appropriato, `UnitNames` o `UnitSymbols` , per utilizzarli. Oppure, se sono richieste solo poche unità SI, possono essere importate con alias di tipo:

```
/// Seconds, the SI unit of time. Type abbreviation for the Microsoft standardized type.
```

```
type [<Measure>] s = Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols.s
```

Alcuni utenti tendono a fare quanto segue, cosa che **non dovrebbe essere fatta** ogni volta che una definizione è già disponibile:

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

La differenza diventa evidente quando si interfaccia con un altro codice che fa riferimento ai tipi SI standard. Il codice che fa riferimento alle unità standard è compatibile, mentre il codice che definisce il proprio tipo è incompatibile con qualsiasi codice che non utilizza la sua specifica definizione.

Pertanto, utilizzare sempre i tipi standard per le unità SI. Non importa se ti riferisci a `UnitNames` o `UnitSymbols` , poiché i nomi equivalenti all'interno di questi due si riferiscono allo stesso tipo:

```
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>
```

Leggi Unità di misura online: https://riptutorial.com/it/fsharp/topic/1055/unita-di-misura

Capitolo 32: Usando F #, WPF, FsXaml, un Menu e una finestra di dialogo

introduzione

L'obiettivo qui è quello di creare una semplice applicazione in F # utilizzando Windows Presentation Foundation (WPF) con menu e finestre di dialogo tradizionali. Deriva dalla mia frustrazione nel cercare di guadare attraverso centinaia di sezioni di documentazione, articoli e post relativi a F # e WPF. Per fare qualsiasi cosa con WPF, sembra che tu debba sapere tutto a riguardo. Il mio scopo qui è quello di fornire un possibile metodo, un semplice progetto desktop che può fungere da modello per le tue app.

Examples

Imposta il progetto

Supponiamo che lo stai facendo in Visual Studio 2015 (VS 2015 Community, nel mio caso). Creare un progetto Console vuoto in VS. Nel progetto | Le proprietà cambiano il tipo di output nell'applicazione Windows.

Quindi, utilizzare NuGet per aggiungere FsXaml.Wpf al progetto; questo pacchetto è stato creato dalla stimabile Reed Copsey, Jr., e semplifica enormemente l'utilizzo di WPF da F #. Durante l'installazione, aggiungerà un numero di altri assembly WPF, quindi non sarà necessario. Esistono altri pacchetti simili a FsXaml, ma uno dei miei obiettivi era quello di mantenere il numero di strumenti il più piccolo possibile al fine di rendere il progetto generale il più semplice e più attuabile possibile.

Inoltre, aggiungere UIAutomationTypes come riferimento; viene fornito come parte di .NET.

Aggiungi la "Business Logic"

Presumibilmente, il tuo programma farà qualcosa. Aggiungi il tuo codice di lavoro al progetto al posto di Program.fs. In questo caso, il nostro compito è disegnare curve spirograph su una Window Canvas. Questo si ottiene usando Spirograph.fs, sotto.

```
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 è il primo file F # nell'ordine di compilazione, quindi contiene le definizioni dei tipi di cui abbiamo bisogno. Il suo compito è disegnare uno spirograph sulla finestra principale Canvas in base ai parametri immessi in una finestra di dialogo. Dato che ci sono molti riferimenti su come disegnare uno spirograph, non entreremo in questo qui.

Crea la finestra principale in XAML

Devi creare un file XAML che definisca la finestra principale che contiene il nostro menu e lo spazio di disegno. Ecco il codice XAML 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>
```

I commenti non sono in genere inclusi nei file XAML, che penso sia un errore. Ho aggiunto alcuni commenti a tutti i file XAML in questo progetto. Non asserisco che siano i migliori commenti mai scritti, ma almeno mostrano come un commento dovrebbe essere formattato. Tieni presente che i commenti nidificati non sono consentiti in XAML.

Crea la finestra di dialogo in XAML e F

Il file XAML per i parametri dello spirografo è sotto. Comprende tre caselle di testo per i parametri dello spirografo e un gruppo di tre pulsanti radio per il colore. Quando diamo ai pulsanti radio lo stesso nome di gruppo - come abbiamo qui - WPF gestisce la commutazione on / off quando ne viene selezionato uno.

```
<!-- 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>
```

Ora aggiungiamo il codice per Dialog.Box. Per convenzione, il codice utilizzato per gestire l'interfaccia della finestra di dialogo con il resto del programma è denominato XXX.xaml.fs, in cui il file XAML associato è denominato XXX.xaml.

```
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
    ()
```

Gran parte del codice qui è dedicato a garantire che i parametri dello spirografo in Spirograph.fs corrispondano a quelli mostrati in questa finestra di dialogo. Si noti che non vi è alcun controllo degli errori: se si inserisce un punto mobile per gli interi previsti nei primi due campi dei parametri, il programma si bloccherà. Quindi, per favore aggiungi il controllo degli errori nel tuo sforzo.

Si noti inoltre che i campi di immissione dei parametri sono disabilitati con il colore casuale selezionato nei pulsanti di opzione. È qui solo per mostrare come può essere fatto.

Per spostare i dati avanti e indietro tra la finestra di dialogo e il programma, utilizzo System.Windows.Element.FindName () per trovare il controllo appropriato, lanciarlo sul controllo che dovrebbe essere, e quindi ottenere le impostazioni rilevanti dal Controllo. La maggior parte degli altri programmi di esempio utilizza l'associazione dati. Non l'ho fatto per due motivi: in primo luogo, non riuscivo a capire come farlo funzionare, e in secondo luogo, quando non ha funzionato, non ho ricevuto alcun messaggio di errore di alcun tipo. Forse qualcuno che visita questo sito su StackOverflow può dirmi come utilizzare l'associazione dati senza includere un intero nuovo set di pacchetti NuGet.

Aggiungi il codice dietro per MainWindow.xaml

```
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
```

Non c'è molto da fare qui: apriamo la finestra di dialogo Parametri quando richiesto e abbiamo la possibilità di ridisegnare lo spirografo con qualunque sia il parametro corrente.

Aggiungi App.xaml e App.xaml.fs per legare tutto insieme

```
<!-- 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>
```

Ecco il codice dietro:

```
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 è qui tutto gasplate, principalmente per mostrare dove possono essere dichiarate le risorse dell'applicazione, come icone, grafici o file esterni. La compagna App.xaml.fs riunisce il modello e la finestra principale, dimensiona la finestra principale a due terzi delle dimensioni dello schermo disponibili e la esegue.

Quando si crea questo, ricordarsi di assicurarsi che la proprietà Build per ciascun file xaml sia impostata su Risorsa. Quindi è possibile eseguire il debugger o compilare un file exe. Si noti che non è possibile eseguire ciò utilizzando l'interprete F #: il pacchetto FsXaml e l'interprete non sono compatibili.

Ecco qua. Spero che tu possa utilizzare questo come punto di partenza per le tue applicazioni e, in tal modo, puoi estendere la tua conoscenza oltre ciò che viene mostrato qui. Eventuali commenti e suggerimenti saranno apprezzati.

Leggi Usando F #, WPF, FsXaml, un Menu e una finestra di dialogo online: https://riptutorial.com/it/fsharp/topic/9145/usando-f-sharp--wpf--fsxaml--un-menu-e-una-finestra-di-dialogo

Capitolo 33: Valutazione pigra

Examples

Introduzione alla valutazione pigra

La maggior parte dei linguaggi di programmazione, incluso F #, valuta immediatamente i calcoli in base a un modello chiamato Strict Evaluation. Tuttavia, in Lazy Evaluation, i calcoli non vengono valutati fino a quando non sono necessari. F # ci consente di utilizzare la valutazione lazy attraverso sia la parola chiave `lazy` che le `sequences` .

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

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

Inoltre, quando si utilizza Lazy Evaluation, i risultati del calcolo vengono memorizzati nella cache, quindi se si forza il risultato dopo la prima istanza di forzatura, l'espressione stessa non verrà valutata di nuovo

```
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()
```

Introduzione alla valutazione pigra in F

F #, come la maggior parte dei linguaggi di programmazione, utilizza la valutazione rigorosa per impostazione predefinita. In Strict Evaluation, i calcoli vengono eseguiti immediatamente. Al contrario, Lazy Evaluation, rimuove l'esecuzione dei calcoli fino a quando i loro risultati sono necessari. Inoltre, i risultati di un calcolo in Lazy Evaluation vengono memorizzati nella cache, ovviando così alla necessità di rivalutare un'espressione.

Possiamo utilizzare la valutazione Lazy in F # tramite la parola chiave `lazy` e `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()
```

Leggi Valutazione pigra online: https://riptutorial.com/it/fsharp/topic/3682/valutazione-pigra

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con F #	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	1: codice WPF F # dietro l'applicazione con FsXaml	Bent Tranberg
3	Classi	asibahi, inzi, RamenChef, Tomasz Maczyński
4	elenchi	asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj
5	Estensioni di tipo e modulo	Jono Job
6	F # su .NET Core	Boggin, Joel Martinez
7	Flussi di lavoro di sequenza	Jwosty
8	Folds	Jean-Claude Colette, Zaid Ajaj
9	funzioni	asibahi, Julien Pires, rmunn, ronilk
10	Generics	Jake Lishman
11	Il tipo "unità"	4444, Abel, Jake Lishman, rmunn
12	Implementazione del modello di progettazione in F #	FuleSnabel, Ringil
13	Introduzione a WPF in F #	Funk
14	Memoizzazione	Jean-Claude Colette, Julien Pires, Ringil
15	Modelli attivi	Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
16	monadi	FuleSnabel
17	operatori	FuleSnabel

18	Parametri di tipo staticamente risolti	Maslow
19	Pattern Matching	asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene
20	Porting C # a F #	jdphenix, marklam, RamenChef
21	Processore di cassette postali	Honza Brestan
22	Records	eirik, goric, Ringil
23	Riflessione	FuleSnabel
24	Sequenza	Foggy Finder, inzi, James McCalden, Julien Pires, s952163
25	Sindacati discriminati	chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen, Vandroiy, VillasV
26	stringhe	FireAlkazar, Julien Pires
27	Suggerimenti e trucchi per le prestazioni F #	FuleSnabel, Paul Westcott, Ringil, s952163
28	tipi	Cedric Royer-Bertrand, FuleSnabel, Julien Pires
29	Tipi di opzioni	asibahi, chillitom, FuleSnabel
30	Tipo Provider	GregC, jdphenix, Joel Martinez
31	Unità di misura	asibahi, goric, GregC, Vandroiy
32	Usando F #, WPF, FsXaml, un Menu e una finestra di dialogo	Bob McCrory, Goswin
33	Valutazione pigra	inzi