EBook Gratis

APRENDIZAJE F#

Free unaffiliated eBook created from
Stack Overflow contributors.

#f#

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con F

Observaciones

F # es un lenguaje "funcional primero". Puedes aprender sobre todos los diferentes tipos de expresiones , junto con las funciones .

El compilador F #, que es de código abierto , compila sus programas en IL, lo que significa que puede usar el código F # desde cualquier lenguaje compatible con .NET como C # ; y ejecútelo en Mono, .NET Core o .NET Framework en Windows.

Versiones

Versión	Fecha de lanzamiento
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

Instalación o configuración

Windows

Si tiene instalado Visual Studio (cualquier versión incluyendo Express y community), F # ya debería estar incluido. Simplemente elija F # como el idioma cuando cree un nuevo proyecto. O vea http://fsharp.org/use/windows/ para más opciones.

OS X

Xamarin Studio soporta F #. Alternativamente, puede usar VS Code para OS X , que es un editor multiplataforma de Microsoft.
Una vez hecho esto con la instalación de VS Code, inicie `VS Code Quick Open` (`Ctrl` + `P`) y luego ejecute `ext install Ionide-fsharp`

También puede considerar Visual Studio para Mac .

Hay otras alternativas descritas aquí .

Linux

Instale los paquetes `mono-complete` y `fsharp` través del `fsharp` paquetes de su distribución (Apt, Yum, etc.). Para una buena experiencia de edición, use el código de Visual Studio e instale el `ionide-fsharp` , o use Atom e instale el `ionide-installer` . Vea http://fsharp.org/use/linux/ para más opciones.

¡Hola Mundo!

Este es el código para un proyecto de consola simple, que imprime "¡Hola, mundo!" A STDOUT, y sale con un código de salida de `0`

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

Ejemplo de desglose línea por línea:

- `[<EntryPoint>]` - Un atributo .net que marca "el método que usas para establecer el punto de entrada" de tu programa (fuente).
- `let main argv` : esto define una función llamada `main` con un solo parámetro `argv` . Debido a que este es el punto de entrada del programa, `argv` será una matriz de cadenas. El contenido de la matriz son los argumentos que se pasaron al programa cuando se ejecutó.
- `printfn "Hello, World!"` - la función `printfn` genera la cadena ** pasada como su primer argumento, también anexando una nueva línea.
- `0` funciones `0` - F # siempre devuelven un valor, y el valor devuelto es el resultado de la última expresión en la función. Poner `0` como la última línea significa que la función siempre devolverá cero (un número entero).

** En realidad, esto *no es* una cadena a pesar de que parece una. En realidad, es un TextWriterFormat , que opcionalmente permite el uso de argumentos comprobados de tipo estático. Pero para el propósito de un ejemplo de "hola mundo" se puede pensar que es una cuerda.

F # interactivo

F # Interactive, es un entorno REPL que le permite ejecutar el código F #, una línea a la vez.

Si ha instalado Visual Studio con F #, puede ejecutar F # Interactive en la consola escribiendo `"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"` . En Linux u OS X, el comando es `fsharpi` , que debe estar en `/usr/bin` o en `/usr/local/bin` dependiendo de cómo instaló F #. De cualquier manera, el comando debe estar en su `PATH` para que pueda sólo tiene

que escribir `fsharpi` .

Ejemplo de uso interactivo de 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
```

Utilice `#help;;` por ayuda

Tenga en cuenta el uso de `;;` para decirle a la REPL que ejecute cualquier comando previamente escrito.

Lea Empezando con F # en línea: https://riptutorial.com/es/fsharp/topic/817/empezando-con-f-sharp

Capítulo 2: 1: Código F # WPF detrás de la aplicación con FsXaml

Introducción

La mayoría de los ejemplos encontrados para la programación F # WPF parecen lidiar con el patrón MVVM, y algunos con MVC, pero casi no hay nada que muestre correctamente cómo ponerse en marcha con el código "bueno".

El código detrás del patrón es muy fácil de usar tanto para la enseñanza como para la experimentación. Se utiliza en numerosos libros de introducción y material de aprendizaje en la web. Es por eso.

Estos ejemplos demostrarán cómo crear un código detrás de la aplicación con ventanas, controles, imágenes e íconos, y más.

Examples

Cree un nuevo código F # WPF detrás de la aplicación.

Crear una aplicación de consola F #.

Cambie el **tipo** de **salida** de la aplicación a la aplicación de *Windows* .

Agregue el paquete **FsXaml** NuGet.

Agregue estos cuatro archivos de origen, en el orden listado aquí.

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.
```

Eliminar el archivo *Program.fs* del proyecto.

Cambie la **acción de compilación** a *recurso* para los dos archivos xaml.

Agregue una referencia al ensamblado .NET **UIAutomationTypes** .

Compilar y ejecutar.

No puede usar el diseñador para agregar controladores de eventos, pero eso no es un problema en absoluto. Simplemente agréguelos manualmente en el código que se encuentra detrás, como se ve con los tres controladores en este ejemplo, incluido el controlador para el botón de prueba.

ACTUALIZACIÓN: Se ha agregado una forma alternativa y probablemente más elegante de

agregar controladores de eventos a FsXaml. Puede agregar el controlador de eventos en XAML, igual que en C # pero debe hacerlo manualmente, y luego anular el miembro correspondiente que aparece en su tipo F #. Yo recomiendo esto

3: Añadir un icono a una ventana

Es una buena idea mantener todos los íconos e imágenes en una o más carpetas.

Haga clic derecho en el proyecto, y use F # Power Tools / New Folder para crear una carpeta llamada Images.

En el disco, coloque su icono en la nueva carpeta *Imágenes* .

De vuelta en Visual Studio, haga clic con el botón derecho en *Imágenes* y use **Agregar / Elemento existente** , luego muestre *Todos los archivos (.)* ** para ver el archivo del icono para que pueda seleccionarlo y luego **agregarlo** .

Seleccione el archivo de icono y establezca su **Acción de** creación en *Recurso* .

En MainWindow.xaml, use el atributo Icono como este. Las líneas circundantes se muestran para el contexto.

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

Antes de ejecutar, realice una reconstrucción y no solo una compilación. Esto se debe a que Visual Studio no siempre coloca el archivo de icono en el ejecutable a menos que lo reconstruya.

Es la ventana, y no la aplicación, la que ahora tiene un icono. Verá el icono en la parte superior izquierda de la ventana en tiempo de ejecución, y lo verá en la barra de tareas. El Administrador de tareas y el Explorador de archivos de Windows no mostrarán este icono, ya que muestran el icono de la aplicación en lugar del icono de la ventana.

4: Añadir icono a la aplicación

Cree un archivo de texto llamado AppIcon.rc, con el siguiente contenido.

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

Necesitará un archivo de icono llamado AppIcon.ico para que esto funcione, pero por supuesto puede ajustar los nombres a su gusto.

Ejecute el siguiente comando.

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

Si no puede encontrar rc.exe en esta ubicación, búsquelo debajo de **C: \ Archivos de programa**

(x86) \ Windows Kits . Si aún no puede encontrarlo, descargue Windows SDK de Microsoft.

Se generará un archivo llamado AppIcon.res.

En Visual Studio, abra las propiedades del proyecto. Seleccione la página de la **aplicación** .

En el cuadro de texto titulado **Archivo de recursos** , escriba *AppIcon.res* (o *Images* \ *AppIcon.res* si lo coloca allí) y luego cierre las propiedades del proyecto para guardar.

Aparecerá un mensaje de error que indica "El archivo de recursos ingresado no existe. Ignórelo. El mensaje de error no volverá a aparecer.

Reconstruir. El ejecutable tendrá un icono de aplicación, y esto se muestra en el Explorador de archivos. Cuando se ejecuta, este icono también aparecerá en el Administrador de tareas.

2: Añadir un control

Agregue estos dos archivos en este orden sobre los archivos de la ventana principal.

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()
```

La **acción** de *creación* para *MyControl.xaml se* debe establecer en *Recurso* .

Por supuesto, más tarde deberá agregar "as this" en la declaración de MyControl, tal como se hizo para la ventana principal.

En el archivo **MainWindow.xaml.fs** , en la clase para MainWindow, agregue esta línea

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

y añadir estas dos líneas en la **DO** -section de la clase de la ventana principal.

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

No puede haber más de un -section **hacer** en una clase, y es probable que se necesite al escribir un montón de código de código subyacente.

Al control se le ha dado un color de fondo verde claro, para que pueda ver fácilmente dónde está.

Tenga en cuenta que el control bloqueará la visualización del botón de la ventana principal. Está más allá del alcance de estos ejemplos enseñarle WPF en general, así que no lo arreglaremos aquí.

Cómo agregar controles desde bibliotecas de terceros

Si agrega controles de bibliotecas de terceros en un proyecto WPF de C #, el archivo XAML normalmente tendrá líneas como esta.

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

Esto quizás no funcione con FsXaml.

El diseñador y el compilador aceptan esa línea, pero probablemente habrá una excepción en el tiempo de ejecución quejándose de que no se encontró el tipo de tercero al leer el XAML.

Intente algo como lo siguiente en su lugar.

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

Este es entonces un ejemplo de un control que depende de lo anterior.

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

La biblioteca utilizada en este ejemplo es el kit de herramientas Extended Wpf, disponible de forma gratuita a través de NuGet o como instalador. Si descarga bibliotecas a través de NuGet, entonces los controles no están disponibles en la Caja de herramientas, pero aún se muestran en el diseñador si los agrega manualmente en XAML, y las propiedades están disponibles en el panel Propiedades.

Lea 1: Código F # WPF detrás de la aplicación con FsXaml en línea: https://riptutorial.com/es/fsharp/topic/9008/1--codigo-f-sharp-wpf-detras-de-la-aplicacion-con-fsxaml

Capítulo 3: Archivos

Examples

Añadir funciones miembro a los registros

```
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
```

Uso básico

```
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
```

Lea Archivos en línea: https://riptutorial.com/es/fsharp/topic/1136/archivos

Capítulo 4: El tipo de "unidad"

Examples

¿De qué sirve una tupla 0?

Una tupla de 2 o una tupla de 3 representa un grupo de elementos relacionados. (Puntos en el espacio 2D, valores RGB de un color, etc.) Una tupla 1 no es muy útil ya que podría reemplazarse fácilmente con un solo `int` .

Un 0-tuple parece aún más inútil ya que no contiene absolutamente *nada* . Sin embargo, tiene propiedades que lo hacen muy útil en lenguajes funcionales como F #. Por ejemplo, el tipo de tupla 0 tiene exactamente *un* valor, generalmente representado como `()` . Todas las tuplas 0 tienen este valor, por lo que es esencialmente un tipo singleton. En la mayoría de los lenguajes de programación funcionales, incluido F #, esto se denomina tipo de `unit` .

Las funciones que devuelven `void` en C # devolverán el tipo de `unit` en F #:

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

Ejecute eso en el intérprete interactivo de F #, y verá:

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

Esto significa que el valor `printResult` es de tipo `unit` y tiene el valor `()` (la tupla vacía, el único valor del tipo de `unit`).

Las funciones también pueden tomar el tipo de `unit` como un parámetro. En F #, las funciones pueden parecer que no están tomando parámetros. Pero, de hecho, están tomando un solo parámetro de tipo `unit` . Esta función:

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

es en realidad equivalente a:

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

Es decir, una función que toma un parámetro de tipo `unit` y devuelve el valor `int` 6. Si observa la firma de tipo que el intérprete interactivo F # imprime cuando define esta función, verá:

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

El hecho de que todas las funciones tomarán al menos un parámetro y devolverá un valor, incluso si ese valor es a veces un valor "inútil" como `()` , significa que la composición de la función es mucho más fácil en F # que en idiomas que no tienen el tipo de `unit` . Pero ese es un tema más

avanzado que veremos más adelante. Por ahora, solo recuerde que cuando ve la `unit` en la firma de una función, o `()` en los parámetros de una función, ese es el tipo de tupla 0 que sirve para decir "Esta función toma, o devuelve, valores no significativos".

Aplazar la ejecución del código

Podemos usar el tipo de `unit` como un argumento de función para definir funciones que no queremos que se ejecuten hasta más adelante. A menudo, esto es útil en tareas en segundo plano asíncronas, cuando el subproceso principal puede querer activar alguna funcionalidad predefinida del subproceso en segundo plano, como moverlo a un nuevo archivo, o si no se debe ejecutar un enlace let de inmediato:

```
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)
```

En el siguiente código, definimos el código para iniciar un "trabajador" que simplemente imprime el valor en el que está trabajando cada 2 segundos. Luego, el trabajador devuelve dos funciones que pueden usarse para controlarlo: una que lo mueve al siguiente valor para trabajar, y otra que lo detiene de funcionar. Estas deben ser funciones, ya que no queremos que sus cuerpos se ejecuten hasta que decidamos hacerlo, de lo contrario, el trabajador pasaría inmediatamente al segundo valor y cerraría sin haber hecho nada.

```
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
```

Entonces podemos empezar un trabajador haciendo

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

y el trabajo comenzará - si estamos en F # interactivo, veremos los mensajes impresos en la consola cada dos segundos. Entonces podemos correr

```
nextValue ()
```

y veremos los mensajes que indican que el valor en el que se está trabajando se ha movido al siguiente.

Cuando es hora de terminar de trabajar, podemos ejecutar el

```
stopOnNextTick ()
```

Función, que imprimirá el mensaje de cierre, luego saldrá.

El tipo de `unit` es importante aquí para indicar "sin entrada": las funciones ya tienen toda la información que necesitan para trabajar, y la persona que llama no tiene permitido cambiar eso.

Lea El tipo de "unidad" en línea: https://riptutorial.com/es/fsharp/topic/2513/el-tipo-de--unidad-

Capítulo 5: Evaluación perezosa

Examples

Introducción perezosa de la evaluación

La mayoría de los lenguajes de programación, incluido F #, evalúan los cálculos inmediatamente de acuerdo con un modelo llamado Evaluación estricta. Sin embargo, en la Evaluación perezosa, los cálculos no se evalúan hasta que son necesarios. F # nos permite usar la evaluación perezosa a través de la palabra clave y las `sequences` `lazy` .

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

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

Además, cuando se usa la Evaluación perezosa, los resultados del cálculo se almacenan en caché, por lo que si forzamos el resultado después de nuestra primera instancia de forzarlo, la expresión en sí no se volverá a evaluar.

```
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()
```

Introducción a la evaluación perezosa en F

F #, como la mayoría de los lenguajes de programación, utiliza la evaluación estricta de forma predeterminada. En la Evaluación Estricta, los cálculos se ejecutan inmediatamente. Por el contrario, Lazy Evaluation, retrasa la ejecución de los cálculos hasta que se necesitan sus resultados. Además, los resultados de un cálculo en Lazy Evaluation se almacenan en caché, lo que evita la necesidad de una reevaluación de una expresión.

Podemos usar la evaluación perezosa en F # a través de la palabra clave `lazy` y las `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()
```

Lea Evaluación perezosa en línea: https://riptutorial.com/es/fsharp/topic/3682/evaluacion-perezosa

Capítulo 6: Extensiones de Tipo y Módulo

Observaciones

En todos los casos, cuando se extienden tipos y módulos, el código de extensión se debe agregar / cargar antes del código que debe llamarse. También debe estar disponible para el código de llamada abriendo / importando los espacios de nombres relevantes.

Examples

Agregando nuevos métodos / propiedades a los tipos existentes

F # permite que las funciones se agreguen como "miembros" a los tipos cuando se definen (por ejemplo, Tipos de registro). Sin embargo, F # también permite agregar nuevos miembros de instancia a *los* tipos *existentes* , incluso los declarados en otros lugares y en otros idiomas .net.

El siguiente ejemplo agrega un nuevo método de instancia `Duplicate` a todas las instancias de `String` .

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

Nota : `this` es un nombre de variable elegido arbitrariamente para usar para referirse a la instancia del tipo que se está extendiendo: `x` funcionaría igual de bien, pero quizás sea menos autodeclarado.

Entonces se puede llamar de las siguientes maneras.

```
// 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
```

Esta funcionalidad es muy similar a los métodos de extensión en C #.

Las nuevas propiedades también se pueden agregar a los tipos existentes de la misma manera. Se convertirán automáticamente en propiedades si el nuevo miembro no toma argumentos.

```
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
```

Añadiendo nuevas funciones estáticas a los tipos existentes.

F # permite que los tipos existentes se amplíen con nuevas funciones estáticas.

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

Esta nueva función se puede invocar así:

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

Esta característica puede significar que, en lugar de tener que crear bibliotecas de funciones "de utilidad", se pueden agregar a los tipos existentes relevantes. Esto puede ser útil para crear más versiones amigables con F # de funciones que permitan funciones como el curry .

```
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
```

Agregar nuevas funciones a los módulos y tipos existentes usando módulos

Los módulos se pueden usar para agregar nuevas funciones a los módulos y tipos existentes.

```
namespace FSharp.Collections

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

La nueva función se puede llamar como si fuera un miembro original de la Lista.

```
open FSharp.Collections

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

Lea Extensiones de Tipo y Módulo en línea:
https://riptutorial.com/es/fsharp/topic/2977/extensiones-de-tipo-y-modulo

Capítulo 7: F # Consejos y trucos de rendimiento

Examples

Usando la recursión de cola para una iteración eficiente

Viniendo de lenguajes imperativos, muchos desarrolladores se preguntan cómo escribir un `for-loop` que salga temprano ya que `F#` no admite `break` , `continue` o `return` . La respuesta en `F#` es usar la recursión de la cola, que es una forma flexible e idiomática de iterar, a la vez que proporciona un rendimiento excelente.

Digamos que queremos implementar `tryFind` for `List` . Si `F#` apoyara el `return` , escribiríamos `tryFind` un poco así:

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

Esto no funciona en `F#` . En su lugar, escribimos la función usando la recursión de cola:

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

La recursión de cola se realiza en `F#` porque cuando el compilador de `F#` detecta que una función es recursiva de cola, vuelve a escribir la recursión en un `while-loop` eficiente `while-loop` . Usando `ILSpy` podemos ver que esto es cierto para nuestro `loop` función:

```
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;
}
```

Aparte de algunas tareas innecesarias (que, con suerte, el JIT-er elimina), esto es esencialmente un bucle eficiente.

Además, la recursión de la cola es idiomática para `F#` ya que nos permite evitar el estado mutable. Considere una función de `sum` que suma todos los elementos en una `List` . Un primer intento obvio sería este:

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

Si reescribimos el bucle en la recursión de la cola, podemos evitar el estado mutable:

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

Por eficiencia, el compilador `F#` transforma en un `while-loop` que usa un estado mutable.

Mide y verifica tus suposiciones de desempeño

Este ejemplo está escrito con `F#` en mente, pero las ideas son aplicables en todos los entornos

La primera regla para optimizar el rendimiento es no basarse en suposiciones; siempre mida y verifique sus suposiciones.

Como no estamos escribiendo el código de máquina directamente, es difícil predecir cómo el compilador y JIT: er transforman su programa en código de máquina. Es por eso que debemos medir el tiempo de ejecución para ver que obtenemos la mejora de rendimiento que esperamos y verificar que el programa real no contenga ningún gasto general oculto.

La verificación es un proceso bastante simple que involucra la ingeniería inversa del ejecutable usando, por ejemplo, herramientas como ILSpy . El JIT: er complica la verificación en que ver el código de máquina real es complicado pero factible. Sin embargo, normalmente el examen del `IL-code` da grandes ganancias.

El problema más difícil es la medición; más difícil porque es difícil configurar situaciones realistas que permitan medir mejoras en el código. Todavía la medición es invaluable.

Analizando funciones simples de F #

Examinemos algunas funciones `F#` simples que acumulan todos los enteros en `1..n` escritos de

varias maneras diferentes. Como el rango es una serie aritmética simple, el resultado se puede calcular directamente, pero para el propósito de este ejemplo, iteramos sobre el rango.

Primero definimos algunas funciones útiles para medir el tiempo que toma una función:

```
// 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` ejecuta una acción repetidamente, necesitamos ejecutar las pruebas durante unos pocos cientos de milisegundos para reducir la varianza.

Luego definimos algunas funciones que acumulan todos los enteros en `1..n` de diferentes maneras.

```
// 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
```

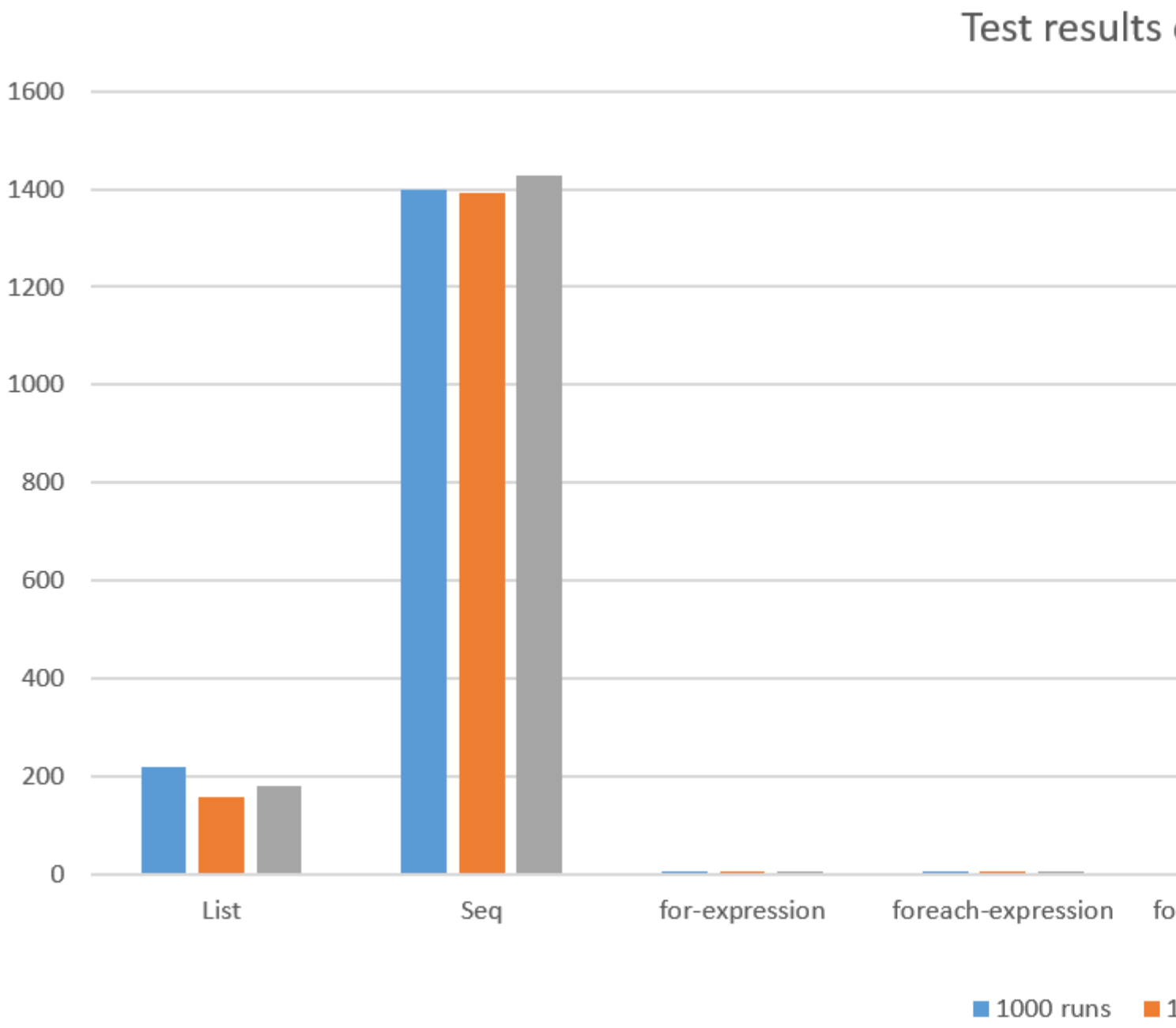
```
// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum

// 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
```

Asumimos que el resultado es el mismo (a excepción de una de las funciones que usa un incremento de `2`), pero existe una diferencia en el rendimiento. Para medir esto se define la siguiente función:

```
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
```

El resultado de la prueba mientras se ejecuta en .NET 4.5.2 x64:



Vemos una diferencia dramática y algunos de los resultados son inesperadamente malos.

Veamos los casos malos:

Lista

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

Lo que sucede aquí es una lista completa que contiene todos los enteros `1..n` se crea y reduce utilizando una suma. Esto debería ser más costoso que solo iterar y acumular en el rango, parece aproximadamente 42 veces más lento que el bucle for.

Además, podemos ver que el GC se ejecutó aproximadamente 100 veces durante la ejecución de

prueba porque el código asignó muchos objetos. Esto también cuesta CPU.

Seq

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

La versión `Seq` no asigna una `List` completa, por lo que es un poco sorprendente que este ~ 270x sea más lento que el bucle for. Además, vemos que el GC ha ejecutado 661x.

`Seq` es ineficiente cuando la cantidad de trabajo por artículo es muy pequeña (en este caso, agregando dos enteros).

El punto es no usar nunca la `Seq` . El punto es medir.

(**edición de manofstick:** `Seq.init` es el culpable de este grave problema de rendimiento. Es mucho más eficaz usar la expresión `{ 0 .. n }` lugar de `Seq.init (n+1) id` . Esto será mucho más eficiente aún cuando este PR se fusiona y se libera. Incluso después del lanzamiento, el `Seq.init ... |> Seq.sum` seguirá siendo lento, pero de manera algo intuitiva, `Seq.init ... |> Seq.map id |> Seq.sum` será bastante rápido. Esto fue para mantener la compatibilidad hacia atrás con la implementación de `Seq.init` , que inicialmente no calcula la `Current` , sino que la envuelve en un objeto `Lazy` , aunque esto también debería funcionar un poco mejor debido a Nota de PR para el editor: disculpe, es una especie de notas entrecortadas, pero no quiero que la gente se desanime Seq cuando la mejora está a la vuelta de la esquina ... *Cuando llegue ese momento, sería bueno actualizar los gráficos que están en esta página.*)

foreach-expresión sobre lista

```
// 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 diferencia entre estas dos funciones es muy sutil, pero la diferencia de rendimiento no lo es, aproximadamente ~ 76x. ¿Por qué? Vamos a aplicar ingeniería inversa al código malo:

```
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` se implementa como un bucle `while` eficiente, pero `for i in [1..n]` se convierte en:

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

Esto significa que primero creamos una `Seq` sobre `1..n` finalmente `ToList` .

Costoso.

foreach-expresión incremento de 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
```

Una vez más, la diferencia entre estas dos funciones es sutil, pero la diferencia de rendimiento es brutal: ~ 25x

Una vez más vamos a ejecutar `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;
}
```

Se crea una `Seq` sobre `1..2..n` y luego `1..2..n` sobre `Seq` usando el enumerador.

Esperábamos que `F#` creara algo como esto:

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

Sin embargo, `F#` compilador `F#` solo es eficaz para bucles en rangos int32 que se incrementan en uno. Para todos los demás casos, recurre a `Operators.OperatorIntrinsics.RangeInt32` . Lo que explicará el próximo resultado sorprendente.

Foreach-expresión más de 64 bits

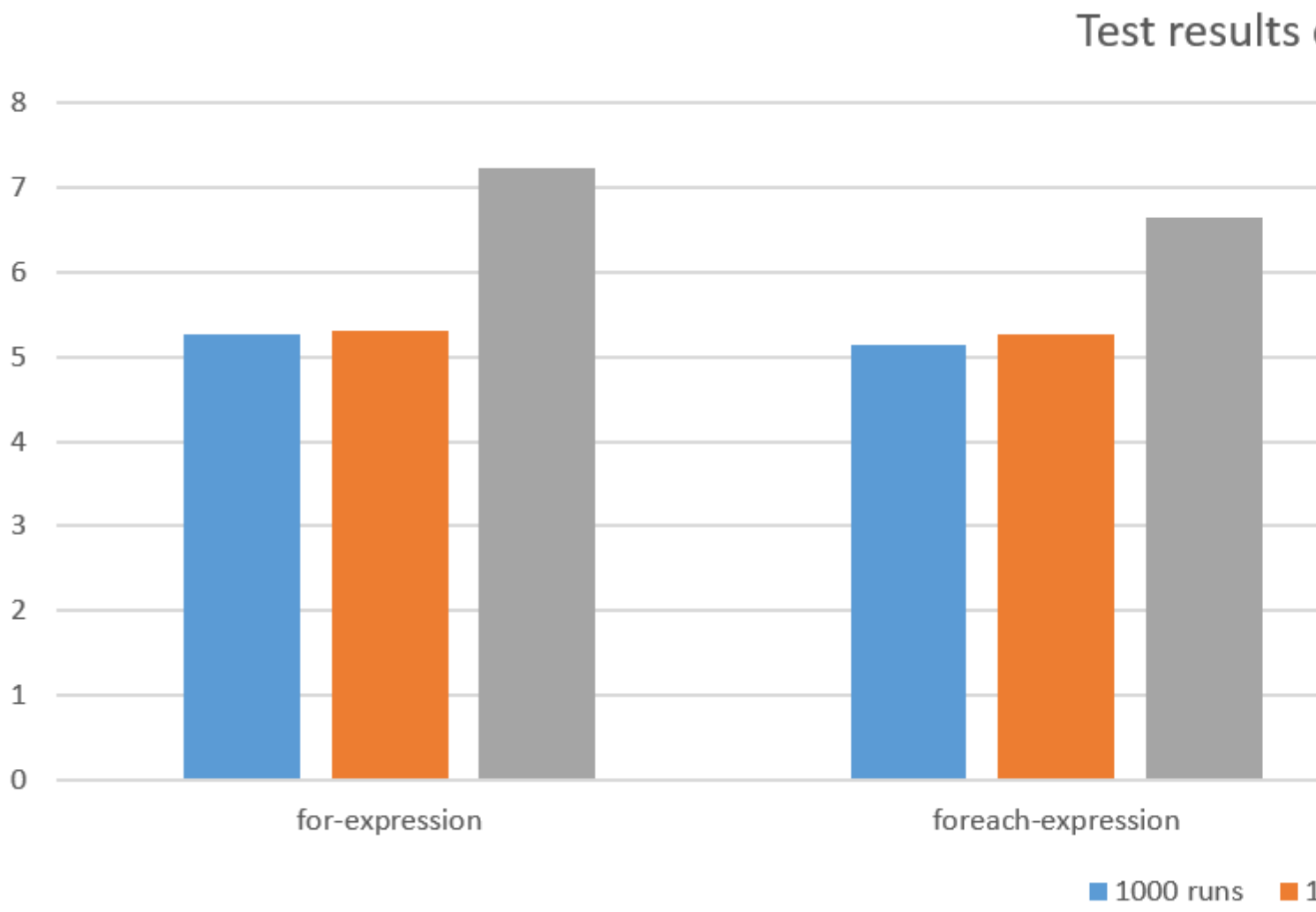
```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int
```

Esto realiza ~ 47x más lento que el bucle for, la única diferencia es que iteramos más de 64 bits enteros. `ILSpy` nos muestra por qué:

```
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#` solo es eficiente para bucles para números `int32` , tiene que usar los `Operators.OperatorIntrinsics.RangeInt64` opositores.OperatorIntrinsics.RangeInt64.

Los otros casos realizan aproximadamente similares:



La razón por la que el rendimiento se degrada para ejecuciones de prueba más grandes es que la sobrecarga de invocar la `action` está creciendo a medida que hacemos menos y menos trabajo en `action` .

El bucle hacia `0` veces puede dar beneficios de rendimiento, ya que podría guardar un registro de CPU, pero en este caso la CPU tiene registros de sobra, por lo que no parece hacer una diferencia.

Conclusión

Medir es importante porque de lo contrario podríamos pensar que todas estas alternativas son equivalentes, pero algunas alternativas son ~ 270x más lentas que otras.

El paso de Verificación involucra ingeniería inversa, el ejecutable nos ayuda a explicar *por qué* obtuvimos o no el rendimiento que esperábamos. Además, la verificación puede ayudarnos a predecir el rendimiento en los casos en los que es demasiado difícil realizar una medición adecuada.

Es difícil predecir el rendimiento allí siempre Medida, siempre Verifique sus suposiciones de rendimiento.

Comparación de diferentes tuberías de datos F

En `F#` hay muchas opciones para crear canales de datos, por ejemplo: `List` , `Seq` y `Array` .

¿Qué canalización de datos es preferible desde el uso de la memoria y la perspectiva del rendimiento?

Para responder a esto, compararemos el rendimiento y el uso de la memoria utilizando diferentes tuberías.

Tubería de datos

Para medir la sobrecarga, utilizaremos un flujo de datos con un bajo costo de CPU por cada artículo procesado:

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

Crearemos tuberías equivalentes para todas las alternativas y las compararemos.

Variaremos el tamaño de `n` pero dejaremos que el número total de trabajos sea el mismo.

Alternativas de la tubería de datos

Vamos a comparar las siguientes alternativas:

1. Código imperativo
2. Array (no perezoso)
3. Lista (No perezoso)
4. LINQ (Lazy pull stream)
5. Seq (corriente de extracción perezosa)
6. Nessos (Lazy pull / push stream)
7. PullStream (flujo de extracción simplista)
8. PushStream (secuencia de empuje simplista)

Aunque no es un flujo de datos, compararemos con el código `Imperative` , ya que coincide más estrechamente con la forma en que la CPU ejecuta el código. Esa debería ser la forma más rápida posible de calcular el resultado que nos permite medir la sobrecarga de rendimiento de las tuberías de datos.

`Array` y la `List` calculan una `Array` / `List` en cada paso, por lo que esperamos una sobrecarga de memoria.

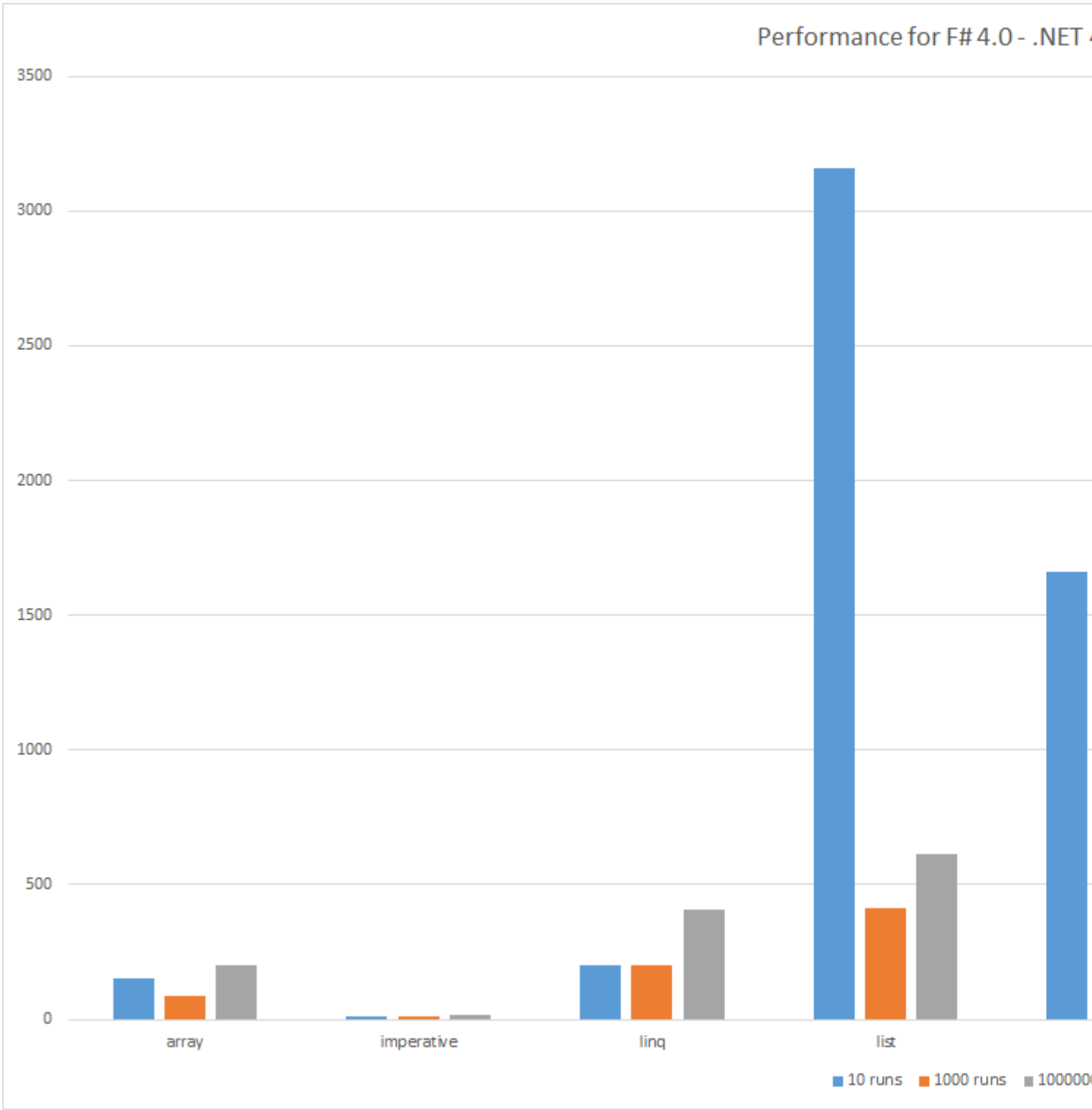
`LINQ`

y `Seq` se basan en `IEnumerable<'T>` que es un flujo perezoso de extracción (extracción significa que el flujo del consumidor está extrayendo datos del productor). Por lo tanto, esperamos que el rendimiento y el uso de la memoria sean idénticos.

`Nessos` es una biblioteca de secuencias de alto rendimiento que admite tanto push & pull (como Java `Stream`).

PullStream y PushStream son implementaciones simplistas de las secuencias de `Pull` & `Push` .

Los resultados de rendimiento se ejecutan en: F # 4.0 - .NET 4.6.1 - x64

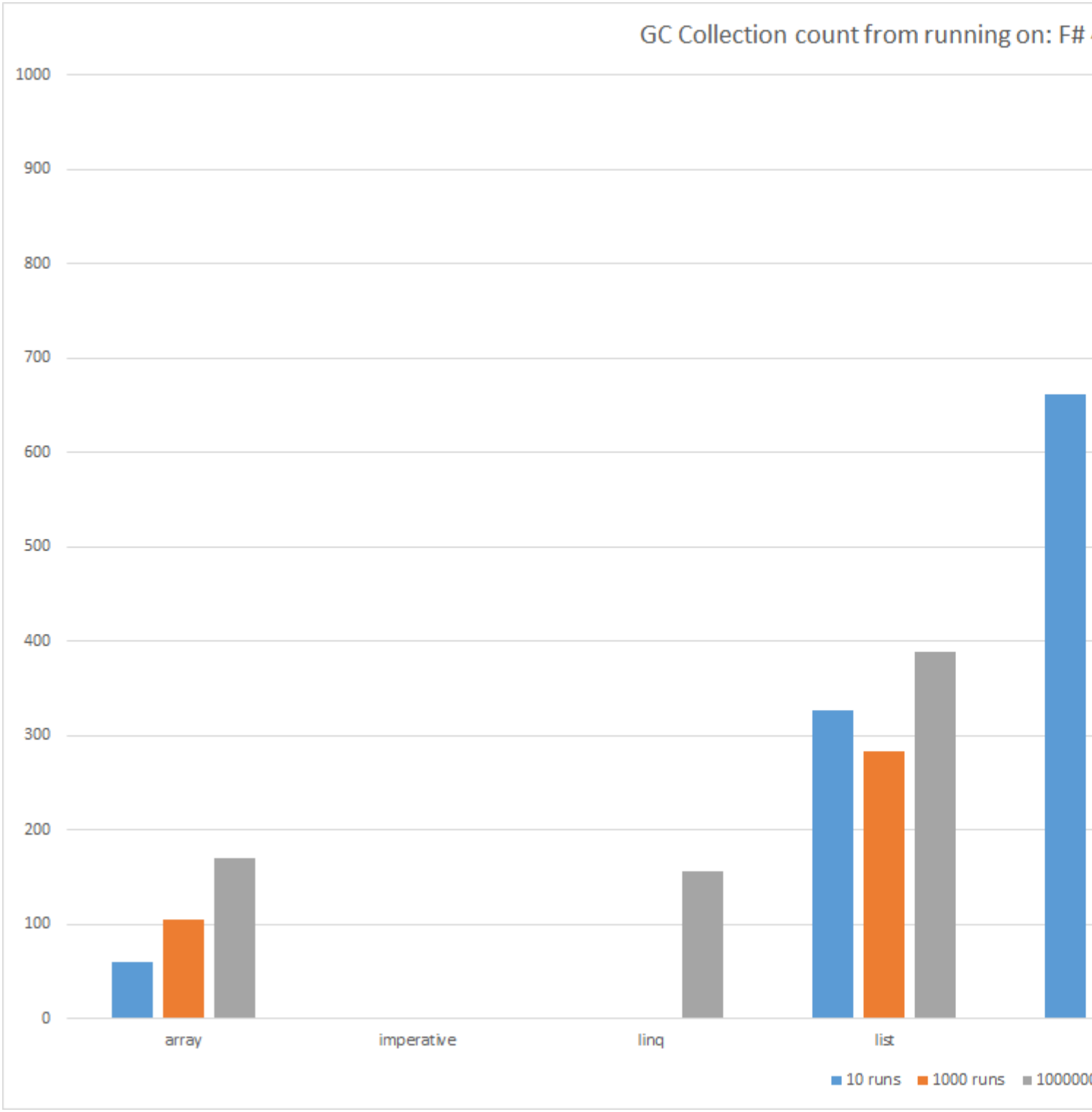


Las barras muestran el tiempo transcurrido, menor es mejor. La cantidad total de trabajo útil es la misma para todas las pruebas, por lo que los resultados son comparables. Esto también significa que pocas ejecuciones implican conjuntos de datos más grandes.

Como es habitual, al medir uno se ven resultados interesantes.

1. `List` rendimiento de la `List` pobre se compara con otras alternativas para grandes conjuntos de datos. Esto puede ser debido a `GC` o mala ubicación de caché.
2. `Array` rendimiento mejor de lo esperado.
3. `LINQ` desempeña mejor que `Seq` , esto es inesperado porque ambos se basan en `IEnumerable<'T>` . Sin embargo, `Seq` internamente se basa en una mejora genérica para todos los algoritmos, mientras que `LINQ` utiliza algoritmos especializados.
4. `Push` realiza mejor que `Pull` . Esto se espera ya que el flujo de datos de inserción tiene menos comprobaciones
5. Las tuberías de datos `Push` simplistas tienen un `Push` comparable al de `Nessos` . Sin embargo, `Nessos` soporta tirón y paralelismo.
6. Para pequeñas tuberías de datos, el rendimiento de `Nessos` degrada debido a la sobrecarga de configuración de las tuberías.
7. Como era de esperar el código `Imperative` realizó el mejor

El recuento de GC Collection se ejecuta en: F # 4.0 - .NET 4.6.1 - x64



Las barras muestran el número total de recuentos de recolección de `GC` durante la prueba, cuanto más bajo, mejor. Esta es una medida de cuántos objetos son creados por la tubería de datos.

Como es habitual, al medir uno se ven resultados interesantes.

1. Se espera que la `List` esté creando más objetos que `Array` porque una `List` es esencialmente una única lista de nodos vinculados. Una matriz es un área de memoria continua.
2. Al observar los números subyacentes, tanto `List` `Array` obligan a las colecciones de 2 generaciones. Este tipo de colección son caras.

3. `Seq` está provocando una sorprendente cantidad de colecciones. Es sorprendentemente incluso peor que la `List` en este sentido.
4. `LINQ` , `Nessos` , `Push` y `Pull` no activan colecciones para pocas ejecuciones. Sin embargo, los objetos se asignan para que el `GC` finalmente tenga que ejecutarse.
5. Como era de esperar, ya que el código `Imperative` no asigna objetos, no se activaron colecciones `GC` .

Conclusión

Todas las tuberías de datos realizan la misma cantidad de trabajo útil en todos los casos de prueba, pero vemos diferencias significativas en el rendimiento y el uso de la memoria entre las distintas tuberías.

Además, observamos que la sobrecarga de las tuberías de datos varía según el tamaño de los datos procesados. Por ejemplo, para tamaños pequeños, `Array` está funcionando bastante bien.

Uno debe tener en cuenta que la cantidad de trabajo realizado en cada paso en la tubería es muy pequeña para medir la sobrecarga. En situaciones "reales", la sobrecarga de `Seq` puede no importar porque el trabajo real lleva más tiempo.

De mayor preocupación son las diferencias de uso de memoria. `GC` no es gratis y es beneficioso para las aplicaciones de ejecución prolongada para mantener baja la presión del `GC` .

Para los desarrolladores de `F#` preocupados por el rendimiento y el uso de la memoria, se recomienda revisar Nessos Streams .

Si necesita un rendimiento `Imperative` código `Imperative` estratégicamente ubicado merece la pena considerarlo.

Finalmente, cuando se trata de rendimiento, no hagas suposiciones. Medir y verificar.

Código fuente 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
```

Lea F # Consejos y trucos de rendimiento en línea: https://riptutorial.com/es/fsharp/topic/3562/f-sharp-consejos-y-trucos-de-rendimiento

Capítulo 8: F # en .NET Core

Examples

Creación de un nuevo proyecto a través de la CLI de dotnet

Una vez que haya instalado las herramientas de .NET CLI, puede crear un nuevo proyecto con el siguiente comando:

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

Esto crea un programa de línea de comandos.

Flujo de trabajo inicial del proyecto

Crear un nuevo proyecto

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

Restaura cualquier paquete listado en project.json

```
dotnet restore
```

Debe escribirse un archivo project.lock.json.

Ejecutar el programa

```
dotnet run
```

Lo anterior compilará el código si es necesario.

La salida del proyecto predeterminado creado por `dotnet new -lf#` contiene lo siguiente:

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

Lea F # en .NET Core en línea: https://riptutorial.com/es/fsharp/topic/4404/f-sharp-en--net-core

Capítulo 9: Flujos de trabajo de secuencia

Examples

rendimiento y rendimiento!

En los flujos de trabajo de secuencia, el `yield` agrega un solo elemento a la secuencia que se está construyendo. (En terminología monádica, es `return` .)

```
> seq { yield 1; yield 2; yield 3 }
val it: seq<int> = seq [1; 2; 3]

> let homogenousTup2ToSeq (a, b) = seq { yield a; yield b }
> tup2Seq ("foo", "bar")
val homogenousTup2ToSeq: 'a * 'a -> seq<'a>
val it: seq<string> = seq ["foo"; "bar"]
```

`yield!` (pronunciado *bang de rendimiento*) inserta todos los elementos de otra secuencia en esta secuencia que se está construyendo. O, en otras palabras, añade una secuencia. (En relación a las mónadas, es `bind`).

```
> seq { yield 1; yield! [10;11;12]; yield 20 }
val it: seq<int> = seq [1; 10; 11; 12; 20]

// Creates a sequence containing the items of seq1 and seq2 in order
> let concat seq1 seq2 = seq { yield! seq1; yield! seq2 }
> concat ['a'..'c'] ['x'..'z']
val concat: seq<'a> -> seq<'a> -> seq<'a>
val it: seq<int> = seq ['a'; 'b'; 'c'; 'x'; 'y'; 'z']
```

Las secuencias creadas por flujos de trabajo de secuencia también son perezosas, lo que significa que los elementos de la secuencia no se evalúan hasta que se necesitan. Algunas formas de forzar elementos incluyen llamar a `Seq.take` (`Seq.take` los primeros n elementos en una secuencia), `Seq.iter` (aplica una función a cada elemento para ejecutar efectos secundarios) o `Seq.toList` (convierte una secuencia en una lista) . ¡Combinar esto con la recursión es donde el `yield!` Realmente empieza a brillar.

```
> 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]
```

para

`for` expresión de secuencia está diseñada para parecerse a su primo más famoso, el imperativo de bucle. Se "enrolla" a través de una secuencia y evalúa el cuerpo de cada iteración en la secuencia que está generando. Al igual que todo lo relacionado con la secuencia, NO es mutable.

```
> 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]
```

Lea Flujos de trabajo de secuencia en línea: https://riptutorial.com/es/fsharp/topic/2785/flujos-de-trabajo-de-secuencia

Capítulo 10: Funciones

Examples

Funciones de más de un parámetro.

En F #, **todas las funciones toman exactamente un parámetro** . Esto parece una declaración extraña, ya que es trivialmente fácil declarar más de un parámetro en una declaración de función:

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

Pero si escribe esa declaración de función en el intérprete interactivo F #, verá que su tipo de firma es:

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

Sin los nombres de los parámetros, esa firma es `int -> int -> int` . El operador `->` es asociativo a la derecha, lo que significa que esta firma es equivalente a `int -> (int -> int)` . En otras palabras, `add` es una función que toma un parámetro `int` , y devuelve **una función que toma un `int` y devuelve `int`** . Intentalo:

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

Sin embargo, también puede llamar a una función como `add` de una manera más "convencional", pasándole directamente dos parámetros, y funcionará como cabría esperar:

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

Esto se aplica a las funciones con tantos parámetros como desee:

```
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
```

Este método de pensar en las funciones de parámetros múltiples como funciones que toman un parámetro y devuelven funciones nuevas (que a su vez pueden tomar un parámetro y devolver funciones nuevas, hasta que llegue a la función final que toma el parámetro final y finalmente

devuelve un resultado) se llama **currying** , en honor al matemático Haskell Curry, famoso por desarrollar el concepto. (Fue inventado por otra persona, pero Curry merece merecidamente la mayor parte del crédito por ello).

Este concepto se usa en F #, y querrás estar familiarizado con él.

Fundamentos de funciones

La mayoría de las funciones en F # se crean con la sintaxis de `let` :

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

Esto define una función llamada `timesTwo` que toma un solo parámetro `x` . Si ejecuta una sesión interactiva de F # (`fsharpi` en OS X y Linux, `fsi.exe` en Windows) y pega esa función (y agrega el `;;` eso le dice a `fsharpi` que evalúe el código que acaba de escribir), verá que responde con:

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

Esto significa que `timesTwo` es una función que toma un solo parámetro `x` de tipo `int` , y devuelve un `int` . Las firmas de funciones a menudo se escriben sin los nombres de los parámetros, por lo que a menudo verá este tipo de función escrito como `int -> int` .

¡Pero espera! ¿Cómo supo F # que `x` era un parámetro entero, ya que nunca especificaste su tipo? Eso se debe a la **inferencia de tipos** . Debido a que en el cuerpo de la función, usted multiplicó `x` por `2` , los tipos de `x` y `2` deben ser iguales. (Como regla general, F # no convertirá implícitamente los valores a diferentes tipos; debe especificar explícitamente las tipografías que desee).

Si desea crear una función que no tome ningún parámetro, esta es la forma **incorrecta** de hacerlo:

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

La forma **correcta** de hacerlo es:

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

Esta función de `hello` tiene el tipo de `unit -> unit` , que se explica en el tipo de "unidad" .

Funciones Curried vs Tupled

Hay dos formas de definir funciones con múltiples parámetros en F #, funciones Curried y funciones 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
```

Todas las funciones definidas desde fuera de F # (como el marco .NET) se utilizan en F # con la forma de Agrupación. La mayoría de las funciones en los módulos de núcleo F # se utilizan con la forma Curried.

La forma al curry se considera F # idiomática, porque permite una aplicación parcial. Ninguno de los dos ejemplos siguientes son posibles con la forma Agrupada.

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

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

La razón detrás de esto es que la función Curried, cuando se llama con un parámetro, devuelve una función. Bienvenido a la programación funcional !!

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

Puedes ver que es exactamente la misma firma.

Sin embargo, al interactuar con otro código .NET, como cuando se escriben bibliotecas, es importante definir las funciones utilizando el formulario Agrupado.

En línea

Alinear le permite reemplazar una llamada a una función con el cuerpo de la función.

Esto a veces es útil por razones de rendimiento en una parte crítica del código. Pero la contraparte es que su ensamblaje ocupará mucho espacio ya que el cuerpo de la función se duplica en todos los lugares en que se produjo una llamada. Debe tener cuidado al decidir si desea integrar una función o no.

Una función se puede alinear con la palabra clave en `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"
```

Otro ejemplo con valor local:

```
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
```

Pipa hacia adelante y hacia atrás

Los operadores de tuberías se utilizan para pasar parámetros a una función de una manera simple y elegante. Permite eliminar valores intermedios y hacer que las llamadas a funciones sean más fáciles de leer.

En F #, hay dos operadores de tubería:

- **Adelante** (`|>`): pasando parámetros de izquierda a derecha

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

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

- **Atrás** (`<|`): pasar parámetros de derecha a izquierda

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

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

Aquí hay un ejemplo sin operadores de tubería:

```
// 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
```

Podemos acortar el ejemplo anterior y hacerlo más limpio con el operador de tubería delantera:

```
// 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
```

El resultado de cada función se pasa como un parámetro a la siguiente función.

Si desea pasar varios parámetros al operador de tubería, debe agregar un `|` para cada parámetro adicional y crear una tupla con los parámetros. El operador de tubería F # nativo admite hasta tres parámetros (|||> o <|||).

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

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

Lea Funciones en línea: https://riptutorial.com/es/fsharp/topic/2525/funciones

Capítulo 11: Genéricos

Examples

Reversión de una lista de cualquier tipo.

Para revertir una lista, no es importante el tipo de elementos de la lista, solo el orden en el que están. Este es un candidato perfecto para una función genérica, por lo que se puede usar la misma función reverseal sin importar qué lista se pase.

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

El código no hace suposiciones sobre los tipos de los elementos. El compilador (o F # interactivo) le dirá que la firma de tipo de esta función es `'T list -> 'T list` . La `'T` le dice que es un tipo genérico sin restricciones. También puede ver `'a` lugar de `'T` : la letra no es importante porque es solo un marcador de posición *genérico* . Podemos pasar una `int list` o una `string list` , y ambas funcionarán correctamente, devolviendo una `int list` o una `string list` respectivamente.

Por ejemplo, en F # interactivo:

```
> 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"]
```

Mapeo de una lista en un tipo diferente

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

La firma de esta función es `('a -> 'b) -> 'a list -> 'b list` , que es la más genérica que puede ser. Esto no impide que `'a` a sea del mismo tipo que `'b` , pero también les permite ser diferentes. Aquí puede ver que `'a` tipo que es el parámetro de la función `f` debe coincidir con el tipo del parámetro de `list` . Esta función aún es genérica, pero hay algunas restricciones leves en las entradas: si los tipos no coinciden, habrá un error de compilación.

Ejemplos:

```
> 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'
```

Lea Genéricos en línea: https://riptutorial.com/es/fsharp/topic/7731/genericos

Capítulo 12: Implementación de patrones de diseño en F

Examples

Programación basada en datos en F

Gracias a la inferencia de tipos y la aplicación parcial en la programación basada en datos de `F#` es breve y legible.

Imaginemos que estamos vendiendo seguros de automóviles. Antes de que intentemos venderlo a un cliente, tratamos de determinar si el cliente es un cliente potencial válido para nuestra empresa verificando el sexo y la edad del cliente.

Un simple modelo de cliente:

```
type Sex =
  | Male
  | Female

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

A continuación, queremos definir una lista de exclusión (tabla) para que si un cliente coincide con cualquier fila de la lista de exclusión, el cliente no pueda comprar nuestro seguro de automóvil.

```
//  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
  |]
```

Debido a la inferencia de tipos y la aplicación parcial, la lista de exclusión es flexible pero fácil de entender.

Finalmente, definimos una función que utiliza la lista de exclusión (una tabla) para dividir a los clientes en dos grupos: clientes potenciales y clientes rechazados.

```
// 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 ()
```

Para concluir, definamos algunos clientes y veamos si hay algún cliente potencial para nuestro seguro de automóvil entre ellos:

```
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
```

Esto imprime:

```
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;})|]
```

Lea Implementación de patrones de diseño en F # en línea:
https://riptutorial.com/es/fsharp/topic/3925/implementacion-de-patrones-de-diseno-en-f-sharp

Capítulo 13: Instrumentos de cuerda

Examples

Literales de cuerda

```
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
```

Formato de cadena simple

Hay varias formas de formatear y obtener una cadena como resultado.

La forma de .NET es mediante el uso de `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 # también tiene funciones para formatear cadenas en un estilo C Hay equivalentes para cada una de las funciones .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
```

Usar esas funciones en lugar de las funciones .NET proporciona algunas ventajas:

- Tipo de seguridad
- Aplicacion parcial
- F # soporte de tipo nativo

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/fsharp/topic/1397/instrumentos-de-cuerda

Capítulo 14: Introducción a WPF en F

Introducción

Este tema ilustra cómo explotar la **programación funcional** en una **aplicación WPF** . El primer ejemplo proviene de un post de Māris Krivtežs (ref. Sección de *Comentarios* en la parte inferior). La razón para volver a visitar este proyecto es doble:

1 \ El diseño admite la separación de preocupaciones, mientras que el modelo se mantiene puro y los cambios se propagan de manera funcional.

2 \ La semejanza facilitará la transición a la implementación de Gjallarhorn.

Observaciones

Biblioteca de proyectos de demostración @GitHub

- FSharp.ViewModule (bajo FsXaml)
- Gjallarhorn (muestras de referencia)

Māris Krivtežs escribió dos publicaciones excelentes sobre este tema:

- Aplicación F # XAML: MVVM vs MVC, donde se destacan los pros y los contras de ambos enfoques.

  Siento que ninguno de estos estilos de aplicación XAML se beneficia mucho de la programación funcional. Me imagino que la aplicación ideal consistiría en la vista que produce eventos y los eventos mantienen el estado de vista actual. Toda la lógica de la aplicación debe manejarse filtrando y manipulando los eventos y el modelo de vista, y en la salida debe producir un nuevo modelo de vista que esté vinculado de nuevo a la vista.

- F # XAML: MVVM controlado por evento como revisado en el tema anterior.

Examples

FSharp.ViewModule

Nuestra aplicación de demostración consiste en un marcador. El modelo de puntuación es un registro inmutable. Los eventos del marcador están contenidos en un Tipo de Unión.

```
namespace Score.Model

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

Los cambios se propagan escuchando eventos y actualizando el modelo de vista en consecuencia. En lugar de agregar miembros al tipo de modelo, como en OOP, declaramos un módulo separado para alojar las operaciones permitidas.

```
[<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
```

Nuestro modelo de vista se deriva de `EventViewModelBase<'a>` , que tiene una propiedad `EventStream` de tipo `IObservable<'a>` . En este caso, los eventos a los que queremos suscribirnos son de tipo `ScoringEvent` .

El controlador maneja los eventos de una manera funcional. Su `Score -> ScoringEvent -> Score` distintivo `Score -> ScoringEvent -> Score` nos muestra que, cada vez que ocurre un evento, el valor actual del modelo se transforma en un nuevo valor. Esto permite que nuestro modelo permanezca puro, aunque nuestro modelo de vista no lo sea.

Un `eventHandler` se encarga de mutar el estado de la vista. Al heredar de `EventViewModelBase<'a>` podemos usar `EventValueCommand` y `EventValueCommandChecked` para conectar los eventos a los comandos.

```
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
```

El código detrás del archivo (* .xaml.fs) es donde se pone todo junto, es decir, la función de

actualización (`controller`) se inyecta en `MainViewModel` .

```
namespace Score.Views

open FsXaml

type MainView = XAML<"MainWindow.xaml">

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

El tipo `CompositionRoot` sirve como un contenedor al que se hace referencia en el archivo XAML.

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

No profundizaré más en el archivo XAML ya que es un elemento básico de WPF, el proyecto completo se puede encontrar en GitHub .

Gjallarhorn

Los tipos de núcleo en la biblioteca Gjallarhorn implementan `IObservable<'a>` , lo que hará que la implementación parezca familiar (recuerde la propiedad `EventStream` del ejemplo FSharp.ViewModule). El único cambio real en nuestro modelo es el orden de los argumentos de la función de actualización. Además, ahora usamos el término *Mensaje en* lugar de *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
```

Con el fin de crear una interfaz de usuario con Gjallarhorn, en lugar de crear clases para admitir el enlace de datos, creamos funciones simples denominadas `Component` . En su constructor, el `source` del primer argumento es de tipo `BindingSource` (definido en Gjallarhorn.Bindable), y se utiliza para asignar el modelo a la vista, y los eventos de la vista vuelven a aparecer en los mensajes.

```
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
        ]
```

La implementación actual difiere de la versión FSharp.ViewModule en que dos comandos aún no han implementado correctamente CanExecute. También se enumeran las tuberías de la aplicación.

```
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
```

Se fue con la configuración de la vista desacoplada, combinando el tipo `MainWindow` y la aplicación lógica.

```
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
```

Esto resume los conceptos básicos, para obtener información adicional y un ejemplo más detallado, consulte la publicación de Reed Copsey . El proyecto de los *árboles de Navidad* destaca algunos beneficios de este enfoque:

- Redimiéndonos efectivamente de la necesidad de (manualmente) copiar el modelo en una colección personalizada de modelos de vista, administrándolos y construyendo manualmente el modelo a partir de los resultados.
- Las actualizaciones dentro de las colecciones se realizan de manera transparente, mientras se mantiene un modelo puro.
- La lógica y la vista están alojadas en dos proyectos diferentes, que enfatizan la separación de preocupaciones.

Lea Introducción a WPF en F # en línea: https://riptutorial.com/es/fsharp/topic/8758/introduccion-a-wpf-en-f-sharp

Capítulo 15: La coincidencia de patrones

Observaciones

La coincidencia de patrones es una característica poderosa de muchos lenguajes funcionales, ya que a menudo permite que las ramas se manejen de manera muy sucinta en comparación con el uso de múltiples declaraciones de estilo `if` / `else if` / `else` . Sin embargo, dadas las opciones suficientes y los guardias de "cuándo" , el ajuste de patrones también puede volverse detallado y difícil de entender de un vistazo.

Cuando esto sucede, los Patrones Activos de F # pueden ser una excelente manera de dar nombres significativos a la lógica coincidente, lo que simplifica el código y también permite la reutilización.

Examples

Opciones de juego

La coincidencia de patrones puede ser útil para manejar las opciones:

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

La coincidencia de patrones comprueba que todo el dominio está cubierto

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

produce una advertencia

> C: \ Archivos de programa (x86) \ Microsoft VS Code \ Untitled-1 (2,7): warning FS0025: El patrón incompleto coincide con esta expresión. Por ejemplo, el valor 'falso' puede indicar un caso que no está cubierto por el patrón (s).

Esto se debe a que no se cubrieron todos los valores bools posibles.

Los bools se pueden enumerar explícitamente pero los ints son más difíciles

de enumerar

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

Aquí usamos el carácter especial _ El _ coincide con todos los demás casos posibles.

El _ puede meterte en problemas

Consideremos un tipo que creamos nosotros mismos se ve así

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

Podríamos escribir una coincidencia con la expesión que se ve así

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

El código anterior tiene sentido. Asumimos que si no está sobrio, debe llamar a un súper, así que usamos _ para indicar que

Más tarde refactorizamos nuestro código para esto.

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

El compilador de F # debe darnos una advertencia y pedirnos que refactoricemos nuestra expresión de coincidencia para que la persona busque atención médica. En cambio, la expresión de coincidencia trata silenciosamente a la persona inconsciente como si solo estuviera borracha. El punto es que debe optar por listar explícitamente los casos cuando sea posible para evitar errores lógicos.

Los casos se evalúan de arriba a abajo y se usa la primera coincidencia.

Uso incorrecto:

En el siguiente fragmento de código, nunca se utilizará la última coincidencia:

```
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"
```

huellas dactilares

> x es algo que no estaba en la lista de arriba

Uso Correcto:

Aquí, tanto `x = 1` como `x = 4` alcanzarán sus casos específicos, mientras que todo lo demás caerá en el caso predeterminado _ :

```
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"
```

huellas dactilares

> x es 4

Cuando los guardias te permiten añadir condicionales arbitrarios.

```
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"
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/fsharp/topic/1335/la-coincidencia-de-patrones

Capítulo 16: Las clases

Examples

Declarando una clase

```
// 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
```

Creando una instancia

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

Lea Las clases en línea: https://riptutorial.com/es/fsharp/topic/3003/las-clases

Capítulo 17: Liza

Sintaxis

- [] // una lista vacía.

  head :: tail // una celda de construcción que contiene un elemento, head y una lista, tail. :: se llama el operador Contras.

  vamos list1 = [1; 2; 3] // Tenga en cuenta el uso de un punto y coma.

  vamos list2 = 0 :: list1 // el resultado es [0; 1; 2; 3]

  let list3 = list1 @ list2 // el resultado es [1; 2; 3; 0; 1; 2; 3]. @ es el operador de adición.

  let list4 = [1..3] // el resultado es [1; 2; 3]

  let list5 = [1..2..10] // el resultado es [1; 3; 5; 7; 9]

  sea list6 = [para i en 1..10 do si i% 2 = 1 entonces el resultado es i] // el resultado es [1; 3; 5; 7; 9]

Examples

Uso básico de la lista

```
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
```

Cálculo de la suma total de números en una lista

Por recursion

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

El ejemplo anterior dice: "Mire la `list` , ¿está vacía? Devuelva 0. De lo contrario, no es una lista vacía. Por lo tanto, podría ser `[1]` , `[1; 2]` , [1; 2; 3] etc. Si la `list` es [1], vincule la variable `head` a `1` y `tail` a `[]` luego ejecute `head + sumTotal tail` .

Ejecución de ejemplo:

```
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
```

Una forma más general de encapsular el patrón anterior es mediante el uso de pliegues funcionales. `sumTotal` convierte en esto:

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

Creando listas

Una forma de crear una lista es colocar los elementos en dos corchetes, separados por punto y coma. Los elementos deben tener el mismo tipo.

Ejemplo:

```
> 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]
```

Cuando una lista no tiene elemento, está vacía. Una lista vacía se puede declarar de la siguiente manera:

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

Otro ejemplo

Para crear una lista de bytes, simplemente para convertir los enteros:

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

También es posible definir listas de funciones, de elementos de un tipo definido previamente, de objetos de una clase, etc.

Ejemplo

```
> 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]
```

Gamas

Para ciertos tipos de elementos (int, float, char, ...), es posible definir una lista por el elemento de inicio y el elemento final, utilizando la siguiente plantilla:

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

Ejemplos:

```
> 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]
```

También puede especificar un paso para ciertos tipos, con el siguiente modelo:

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

Ejemplos:

```
> 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]
```

Generador

Otra forma de crear una lista es generarla automáticamente usando un generador.

Podemos utilizar uno de los siguientes modelos:

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

o

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

Ejemplos

```
> 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]
```

Los operadores

Algunos operadores pueden usarse para construir listas:

Contras operador ::

Este operador :: se usa para agregar un elemento principal a 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]
```

Concatenación

La concatenación de listas se realiza con el operador @.

```
> 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]
```

Lea Liza en línea: https://riptutorial.com/es/fsharp/topic/1268/liza

Capítulo 18: Los operadores

Examples

Cómo componer valores y funciones utilizando operadores comunes.

En Programación Orientada a Objetos una tarea común es componer objetos (valores). En la Programación Funcional es una tarea común componer valores y funciones.

Estamos acostumbrados a componer los valores de nuestra experiencia de otros lenguajes de programación que utilizan operadores como `+` , `-` , `*` , `/` y así sucesivamente.

Composición del valor

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

Como la programación funcional compone funciones y valores, no es sorprendente que haya operadores comunes para la composición de funciones como `>>` , `<<` , `|>` y `<|` .

Composición de funciones

```
// 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')
```

En `F#` tubería hacia adelante se prefiere sobre la tubería inversa porque:

1. La inferencia de tipo (generalmente) fluye de izquierda a derecha, por lo que es natural que los valores y las funciones fluyan de izquierda a derecha también
2. Porque `<|` y `<<` debería tener asociatividad a la derecha, pero en `F#` son asociativas a la izquierda, lo que nos obliga a insertar ()
3. La mezcla de tuberías hacia adelante y hacia atrás generalmente no funciona porque tienen la misma prioridad.

Composición de mónada

Como las mónadas (como la `Option<'T>` o la `List<'T>`) se usan comúnmente en la programación funcional, también hay operadores comunes pero menos conocidos para componer funciones que

trabajan con Mónadas como `>>=` , `>=>` , `<|>` y `<*>` .

```
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
```

Conclusión

Para los nuevos programadores funcionales, la composición de funciones con operadores puede parecer opaca y oscura, pero eso se debe a que el significado de estos operadores no es tan comúnmente conocido como operadores que trabajan con valores. Sin embargo, con un poco de entrenamiento usando `|>` , `>>` , `>>=` y `>=>` vuelve tan natural como usar `+` , `-` , `*` y `/` .

Latebinding en F # utilizando? operador

En un lenguaje de tipo estático como `F#` , trabajamos con tipos conocidos en tiempo de compilación. Consumimos fuentes de datos externas de una manera segura para el tipo utilizando proveedores de tipo.

Sin embargo, ocasionalmente hay necesidad de usar un enlace tardío (como `dynamic` en `C#`). Por ejemplo, cuando se trabaja con documentos `JSON` que no tienen un esquema bien definido.

Para simplificar el trabajo con el enlace tardío, ¿ `F#` proporciona soporte para operadores de búsqueda dinámica `?` y `?<-` .

Ejemplo:

```
// (?) 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
```

Resulta que el soporte de `F#` para el enlace tardío es simple pero flexible.

Lea Los operadores en línea: https://riptutorial.com/es/fsharp/topic/4641/los-operadores

Capítulo 19: Los tipos

Examples

Introducción a los tipos

Los tipos pueden representar varios tipos de cosas. Puede ser un solo dato, un conjunto de datos o una función.

En F #, podemos agrupar los tipos en dos categorías:

- Tipos 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
```

- Tipos .NET
    - Tipo incorporado (int, bool, string, ...)
    - Clases, Estructuras e Interfaces
    - Delegados
    - Arrays

Escriba las abreviaturas

Las abreviaturas de tipo le permiten crear alias en los tipos existentes para darles un sentido más significativo.

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

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

Entonces puedes usar el alias como cualquier otro 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";}
```

Tenga cuidado, los alias no comprueban la consistencia del tipo. Esto significa que dos alias que se dirigen al mismo tipo pueden asignarse entre sí:

```
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";}
```

Los tipos se crean en F # usando la palabra clave de tipo

`F#` usa la palabra clave de `type` para crear diferentes tipos de tipos.

1. Tipo de alias
2. Sindicatos discriminados.
3. Tipos de registro
4. Tipos de interfaz
5. Tipos de clase
6. Tipos de struct

Ejemplos con código `C#` equivalente cuando sea posible:

```
// 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
```

Inferencia de tipos

Reconocimiento

Este ejemplo es una adaptación de este artículo sobre la inferencia de tipos.

¿Qué es el tipo de inferencia?

Inferencia de tipos es el mecanismo que permite al compilador deducir qué tipos se usan y dónde. Este mecanismo se basa en un algoritmo a menudo llamado "Hindley-Milner" o "HM". Vea a continuación algunas de las reglas para determinar los tipos de valores simples y de función:

- Mira los literales
- Mira las funciones y otros valores con los que algo interactúa.
- Mira cualquier restricción de tipo explícito
- Si no hay restricciones en ninguna parte, generalice automáticamente a tipos genéricos

Mira los literales

El compilador puede deducir tipos mirando los literales. Si el literal es un int y usted le agrega una "x", entonces "x" también debe ser un int. Pero si el literal es un flotador y le está agregando una "x", entonces "x" también debe ser un flotador.

Aquí hay unos ejemplos:

```
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"
```

Mira las funciones y otros valores con los que interactúa.

Si no hay literales en ninguna parte, el compilador intenta resolver los tipos analizando las funciones y otros valores con los que interactúan.

```
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
```

Mira cualquier restricción de tipo explícito o anotaciones

Si se especifican restricciones de tipo explícitas o anotaciones, entonces el compilador las usará.

```
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
```

Generalización automática

Si después de todo esto, no se encuentran restricciones, el compilador simplemente hace que los tipos sean genéricos.

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

Cosas que pueden ir mal con la inferencia de tipos.

La inferencia de tipos no es perfecta, por desgracia. A veces el compilador simplemente no tiene ni idea de qué hacer. Nuevamente, entender lo que está sucediendo realmente te ayudará a mantener la calma en lugar de querer matar al compilador. Estas son algunas de las principales razones de los errores de tipo:

- Declaraciones fuera de orden
- No hay suficiente información
- Métodos sobrecargados

Declaraciones fuera de orden

Una regla básica es que debe declarar las funciones antes de que se utilicen.

Este código falla:

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

Pero esto está bien:

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

Declaraciones recursivas o simultáneas.

Una variante del problema "fuera de servicio" ocurre con funciones recursivas o definiciones que tienen que referirse entre sí. Ninguna cantidad de reordenamiento ayudará en este caso, necesitamos usar palabras clave adicionales para ayudar al compilador.

Cuando se está compilando una función, el identificador de la función no está disponible para el cuerpo. Entonces, si define una función recursiva simple, obtendrá un error del compilador. La solución es agregar la palabra clave "rec" como parte de la definición de la función. Por ejemplo:

```
// 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
```

Aquí está la versión fija con "rec fib" agregada para indicar que es recursiva:

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

No hay suficiente información

A veces, el compilador simplemente no tiene suficiente información para determinar un tipo. En el siguiente ejemplo, el compilador no sabe en qué tipo se supone que funciona el método Length. Pero tampoco puede hacer que sea genérico, así que se queja.

```
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 ...
```

Este tipo de error se puede solucionar con anotaciones explícitas.

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

Métodos sobrecargados

Al llamar a una clase o método externo en .NET, a menudo recibirá errores debido a la

sobrecarga.

En muchos casos, como el siguiente ejemplo de concat, tendrá que anotar explícitamente los parámetros de la función externa para que el compilador sepa a qué método sobrecargado llamar.

```
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 veces, los métodos sobrecargados tienen diferentes nombres de argumento, en cuyo caso también puede dar una pista al compilador al nombrar los argumentos. Aquí hay un ejemplo para el constructor StreamReader.

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

Lea Los tipos en línea: https://riptutorial.com/es/fsharp/topic/3559/los-tipos

Capítulo 20: Memorización

Examples

Memorización simple

La memorización consiste en resultados de la función de almacenamiento en caché para evitar calcular el mismo resultado varias veces. Esto es útil cuando se trabaja con funciones que realizan cálculos costosos.

Podemos usar una función factorial simple como ejemplo:

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

    innerLoop index 1
```

Llamar a esta función varias veces con el mismo parámetro resulta en el mismo cálculo una y otra vez.

La memorización nos ayudará a almacenar en caché el resultado factorial y devolverlo si se pasa nuevamente el mismo parámetro.

Aquí hay una implementación de memoria simple:

```
// 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 función de `memoization` simplemente toma una función como parámetro y devuelve una función con la misma firma. Podría ver esto en la firma del método `f:('a -> 'b) -> ('a -> 'b)` . De esta manera, puede utilizar la memorización de la misma manera que si estuviera llamando al método

factorial.

Las llamadas a `printfn` son para mostrar lo que sucede cuando llamamos a la función varias veces; Se pueden eliminar de forma segura.

Usar la memorización es fácil:

```
// 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
```

Memoización en una función recursiva.

Utilizando el ejemplo anterior de cálculo del factorial de un número entero, ponga en la tabla hash todos los valores de factorial calculados dentro de la recursión, que no aparecen en la tabla.

Como en el artículo sobre memoización , declaramos una función `f` que acepta un `fact` parámetro de función y un parámetro entero. Esta función `f` , incluye instrucciones para calcular el factorial de `n` desde `fact(n-1)` .

Esto permite manejar la recursión mediante la función devuelta por `memorec` y no por el `fact` mismo y posiblemente detener el cálculo si el valor de `fact(n-1)` ya se ha calculado y se encuentra en la tabla 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
```

Resultado:

```
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
```

Lea Memorización en línea: https://riptutorial.com/es/fsharp/topic/2698/memorizacion

Capítulo 21: Mónadas

Examples

La comprensión de las mónadas viene de la práctica

Este tema está dirigido a desarrolladores de F # intermedios a avanzados

"¿Qué son las mónadas?" Es una pregunta común. Esto es fácil de responder, pero al igual que en la guía Hitchhikers a galaxia, nos damos cuenta de que no entendemos la respuesta porque no sabíamos qué era lo que preguntábamos.

Muchos creen que la forma de entender las Mónadas es practicándolas. Como programadores, normalmente no nos importa la base matemática de lo que son los Principios de Substitución de Liskov, los subtipos o subclases. Al usar estas ideas hemos adquirido una intuición de lo que representan. Lo mismo es cierto para las mónadas.

Para ayudarlo a comenzar con Monads, este ejemplo muestra cómo construir una biblioteca de Combinator de Parser Parser . Esto podría ayudarlo a comenzar, pero la comprensión vendrá de la escritura de su propia biblioteca Monadic.

Suficiente prosa, tiempo para codificar.

El tipo de analizador:

```
// 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 esta definición de un analizador definimos algunas funciones fundamentales del analizador

```
// 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` es una función que, dada una función `sat` , produce un analizador que tiene éxito si no hemos superado la `EOS` y el carácter en la posición actual pasa la función `sat` . Con el uso de `satisfy` creamos una serie de analizadores de caracteres útiles.

Ejecutando esto en 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)
```

Tenemos algunos analizadores fundamentales en su lugar. Los combinaremos en analizadores más potentes utilizando las funciones del combinador de analizador

```
// '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)
```

Los nombres y las firmas no se eligen de forma arbitraria, pero no profundizaremos en esto, en cambio, veamos cómo usamos `bind` para combinar el analizador en otros más complejos:

```
> 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)
```

Lo que esto nos muestra es que `bind` nos permite combinar dos analizadores en un analizador más complejo. Como el resultado de `bind` es un analizador que a su vez se puede combinar de nuevo.

```
> 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` será la forma fundamental en que combinamos los analizadores aunque definiremos funciones de ayuda para simplificar la sintaxis.

Una de las cosas que puede simplificar la sintaxis son las expresiones de cálculo . Son fáciles de definir:

```
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)
```

Esto es 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)
```

Otro combinador de analizador fundamental que vamos a utilizar mucho es `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
```

Esto nos permite definir `letterOrDigit` así:

```
> 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 sobre los operadores Infix.

Una preocupación común sobre FP es el uso de operadores infijos inusuales como `>>=` , `>=>` , `<-` y así sucesivamente. Sin embargo, la mayoría no están preocupados por el uso de `+` , `-` , `*` , `/` y `%` , estos son los operadores bien conocidos usados para componer los valores. Sin embargo, una gran parte en la FP es la composición no solo de los valores sino también de las funciones. Para un desarrollador de FP intermedio, los operadores de infijo `>>=` , `>=>` , `<-` son conocidos y deben tener firmas específicas, así como semántica.

Para las funciones que hemos definido hasta ahora, definiríamos los siguientes operadores de infijo utilizados para combinar analizadores:

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

Entonces `>>=` significa `bind` y `<|>` significa `orElse` .

Esto nos permite combinar parsers más sucintos:

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

Para definir algunos combinadores de analizador avanzados que nos permitirán analizar expresiones más complejas, definimos algunos combinadores de analizador más simples:

```
// '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
  }
```

Estamos listos para definir `many` y `sepBy` cuáles son más avanzados, ya que aplican los analizadores de entrada hasta que fallan. Luego `many` y `sepBy` devuelve el resultado agregado:

```
// '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
```

Creación de un analizador de expresión simple

Con las herramientas que creamos ahora podemos definir un analizador para expresiones simples como `1+2*3`

Partimos de la parte inferior mediante la definición de un programa de análisis de números enteros `pint`

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

Intentamos analizar tantos dígitos como podamos, el resultado es una `char list` . Si la lista está vacía, `fail` , de lo contrario, plegaremos los caracteres en un entero.

Prueba de `pint` en FSI:

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

Además, necesitamos analizar los diferentes tipos de operadores utilizados para combinar valores enteros:

```
// 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)
```

Atando todo junto:

```
// '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
  }
```

Ejecutándolo todo en FSI:

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

Conclusión

Al definir `Parser<'T>` , `return_` , `bind` y asegurándose de que obedecen las leyes monádicas, hemos creado un sencillo pero poderoso marco Combinator de Parser Parser.

Las mónadas y los analizadores se juntan porque los analizadores se ejecutan en un estado de analizador. Las mónadas nos permiten combinar analizadores mientras ocultamos el estado del analizador, lo que reduce el desorden y mejora la composibilidad.

El marco que hemos creado es lento y no produce mensajes de error, esto para mantener el código sucinto. FParsec proporciona tanto un rendimiento aceptable como excelentes mensajes de error.

Sin embargo, un solo ejemplo no puede dar a entender las mónadas. Uno tiene que practicar mónadas.

Aquí hay algunos ejemplos de Mónadas que puede intentar implementar para alcanzar su comprensión ganada:

1. Mónada estatal: permite que el estado del entorno oculto se lleve implícitamente
2. Tracer Monad: permite que el estado de rastreo se lleve de forma implícita. Una variante de la mónada estatal.
3. Turtle Monad: una mónada para crear programas Turtle (Logos). Una variante de la mónada estatal.
4. Mónada De Continuación - Una Mónada De Coroutine. Un ejemplo de esto es `async` en F #

Lo mejor para aprender sería crear una aplicación para Monads en un dominio con el que te sientas cómodo. Para mí eso fue parsers.

Código fuente 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
  }
```

Las expresiones de computación proporcionan una sintaxis alternativa a las mónadas en cadena.

Relacionadas con las mónadas están las expresiones de cálculo de `F#` (`CE`). Un programador normalmente implementa un `CE` para proporcionar un enfoque alternativo para encadenar Mónadas, es decir, en lugar de esto:

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

Puedes escribir esto:

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

Ambos estilos son equivalentes y depende de la preferencia del desarrollador cuál elegir.

Para demostrar cómo implementar una `CE` imagine que le gustan todas las trazas para incluir una identificación de correlación. Esta identificación de correlación ayudará a correlacionar las trazas que pertenecen a la misma llamada. Esto es muy útil cuando se tienen archivos de registro que contienen rastros de llamadas concurrentes.

El problema es que es engorroso incluir el ID de correlación como un argumento para todas las funciones. Como Monads permite llevar un estado implícito , definiremos una Log Monad para ocultar el contexto del registro (es decir, el ID de correlación).

Comenzamos por definir un contexto de registro y el tipo de una función que rastrea con el contexto de registro:

```
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 ())
```

También definimos dos funciones de rastreo que se registrarán con el ID de correlación del contexto de registro:

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

`trace` es una `Function<unit>` que significa que se pasará un contexto de registro cuando se invoque. Desde el contexto de registro, recogemos el ID de correlación y lo rastreamos junto con `v`

Además, definimos `bind` y `return_` y como siguen las Leyes de la Mónada, esto forma nuestra Mónada de Log.

```
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
```

Finalmente, definimos `LogBuilder` que nos permitirá usar la sintaxis `CE` para encadenar `Log` Monads.

```
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 ()
```

Ahora podemos definir nuestras funciones que deberían tener el contexto de registro implícito:

```
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
  }
```

Ejecutamos g con:

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

Que imprime:

```
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
```

Observe que el CorrelationId se realiza implícitamente de `run` de `g` a `f` que nos permite el correlato de las entradas de registro durante la resolución de problemas.

`CE` tiene muchas más funciones, pero esto debería ayudarlo a comenzar a definir sus propios `CE` : s.

Código 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
```

Lea Mónadas en línea: https://riptutorial.com/es/fsharp/topic/3320/monadas

Capítulo 22: Parámetros de tipo resueltos estáticamente

Sintaxis

- `s` es una instancia de `^a` que desea aceptar en tiempo de compilación, que puede ser cualquier cosa que implemente los miembros a los que realmente llama usando la sintaxis.
- `^a` es similar a los genéricos que serían `'a` (o `'A` o `'T` por ejemplo) pero estos se resuelven en tiempo de compilación y permiten cualquier cosa que se ajuste a todos los usos solicitados dentro del método. (no se requieren interfaces)

Examples

Uso simple para cualquier cosa que tenga un miembro Length.

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

Clase, interfaz, uso de registros

```
// 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)
```

Llamada de miembro estático

esto aceptará cualquier tipo con un método llamado `GetLength` que no toma nada y devuelve un int:

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

Lea Parámetros de tipo resueltos estáticamente en línea:

https://riptutorial.com/es/fsharp/topic/7228/parametros-de-tipo-resueltos-estaticamente

Capítulo 23: Patrones activos

Examples

Patrones activos simples

Los patrones activos son un tipo especial de coincidencia de patrones donde puede especificar categorías con nombre en las que sus datos pueden caer, y luego usar esas categorías en las declaraciones de `match` .

Para definir un patrón activo que clasifique los números como positivo, negativo o cero:

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

Esto se puede usar en una expresión de coincidencia de patrón:

```
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
```

Patrones activos con parámetros.

Los patrones activos son simplemente funciones simples.

Al igual que las funciones puedes definir parámetros adicionales:

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

Esto se puede utilizar en un patrón que coincida de esta manera:

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

Los patrones activos se pueden usar para validar y transformar argumentos

de funciones

Un uso interesante pero bastante desconocido de los Patrones Activos en `F#` es que pueden usarse para validar y transformar argumentos de funciones.

Considera la forma clásica de hacer validación de argumentos:

```
// 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 ()
```

Normalmente, agregamos código en el método para verificar que los argumentos son correctos. Usando patrones activos en `F#` podemos generalizar esto y declarar la intención en la declaración del argumento.

El siguiente código es equivalente al código anterior:

```
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 ()
```

Para el usuario de la función `f` y `g` no hay diferencia entre las dos versiones diferentes.

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

Una preocupación es si los patrones activos agregan sobrecarga de rendimiento. `ILSpy` para descompilar `f` y `g` para ver si ese es el caso.

```
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();
}
```

Gracias a `inline` Active Patterns no agrega una sobrecarga adicional en comparación con la forma clásica de validar argumentos.

Patrones activos como envolturas de API .NET

Los patrones activos se pueden usar para hacer que las API de .NET se sientan más naturales, especialmente aquellas que usan un parámetro de salida para devolver algo más que el valor de retorno de la función.

Por ejemplo, normalmente llamaría al método `System.Int32.TryParse` siguiente manera:

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

Puedes mejorar esto un poco usando la coincidencia de patrones:

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

Sin embargo, también podemos definir el siguiente patrón activo que envuelve la función `System.Int32.TryParse` :

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

Ahora podemos hacer lo siguiente:

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

Otro buen candidato para ser envuelto en un patrón activo son las expresiones regulares de la API:

```
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"
```

Patrones activos parciales y completos

Hay dos tipos de patrones activos que difieren un poco en el uso: Completo y Parcial.

Se pueden usar patrones activos completos cuando se pueden enumerar todos los resultados, como "¿es un número impar o par?"

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

Observe que la definición de patrón activo enumera los casos posibles y nada más, y el cuerpo devuelve uno de los casos enumerados. Cuando lo usas en una expresión de coincidencia, esta es una coincidencia completa:

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

Esto es útil cuando desea desglosar el espacio de entrada en categorías conocidas que lo cubren por completo.

Los patrones activos parciales, por otro lado, le permiten ignorar explícitamente algunos resultados posibles al devolver una `option` . Su definición utiliza un caso especial de _ para el caso no coincidente.

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

De esta manera podemos hacer coincidir incluso cuando algunos casos no pueden ser manejados por nuestra función de análisis.

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

Los patrones activos parciales se pueden usar como una forma de prueba, ya sea que la entrada

caiga en una categoría específica en el espacio de entrada al ignorar otras opciones.

Lea Patrones activos en línea: https://riptutorial.com/es/fsharp/topic/962/patrones-activos

Capítulo 24: Pliegues

Examples

Introducción a los pliegues, con un puñado de ejemplos.

Los pliegues son funciones (de orden superior) usadas con secuencias de elementos. Se colapsan `seq<'a>` en `'b` donde `'b` es cualquier tipo (posiblemente todavía `'a`). Esto es un poco abstracto, así que veamos ejemplos prácticos concretos.

Cálculo de la suma de todos los números.

En este ejemplo, `'a` es un `int` . Tenemos una lista de números y queremos calcular la suma de todos los números. Para sumar los números de la lista `[1; 2; 3]` escribimos

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

Permítanme explicar, porque estamos tratando con listas, usamos `fold` en el módulo `List` , por `List.fold` tanto `List.fold` . El primer argumento que `fold` quita es una función binaria, la **carpeta** . El segundo argumento es el **valor inicial** . `fold` comienza a plegar la lista aplicando consecutivamente la función de carpeta a los elementos de la lista comenzando con el valor inicial y el primer elemento. Si la lista está vacía, se devuelve el valor inicial!

La vista general esquemática de un ejemplo de ejecución se ve así:

```
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 función `List.sum` es aproximadamente `List.fold add LanguagePrimitives.GenericZero` donde el cero genérico lo hace compatible con enteros, flotadores, enteros grandes, etc.

Contando elementos en una lista (`count` implementación)

Esto se hace casi de la misma manera que antes, pero al ignorar el valor real del elemento en la lista y en su lugar agregar 1.

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

Esto también se puede hacer así:

```
[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
```

Para que puedas definir el `count` siguiente manera:

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

Encontrando el máximo de la lista.

Esta vez usaremos `List.reduce` que es como `List.fold` pero sin un valor inicial, como en este caso donde no sabemos cuál es el tipo de los valores que comparamos:

```
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
```

Encontrar el mínimo de una lista.

Al igual que al encontrar el máximo, la carpeta es diferente

```
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
```

Listas de concatenacion

Aquí estamos tomando la lista de listas La función de carpeta es el operador `@`

```
// [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]
```

O podría usar operadores binarios como su función de carpeta:

```
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...
```

Cálculo del factorial de un número.

La misma idea que al sumar los números, pero ahora los multiplicamos. si queremos el factorial de `n` , multiplicamos todos los elementos de la lista `[1 .. n]` . El código se convierte en:

```
// 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]
```

Implementando `forall` , `exists` y `contains`

la función `forall` comprueba si todos los elementos en una secuencia satisfacen una condición. `exists` verificación si al menos un elemento en la lista cumple con la condición. Primero necesitamos saber cómo contraer una lista de valores `bool` . Bueno, ¡usamos pliegues por supuesto! Los operadores booleanos serán nuestras funciones de carpeta.

Para verificar si todos los elementos en una lista son `true` , los `&&` con la función `&&` con `true` como valor inicial.

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

Del mismo modo, para verificar si un elemento es `true` en una lista booleanos, lo colapsamos con el `||` operador con `false` como valor inicial:

```
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
```

Volver a `forall` y `exists` . Aquí tomamos una lista de cualquier tipo, usamos la condición para transformar todos los elementos a valores booleanos y luego la colapsamos:

```
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
```

Para comprobar si todos los elementos en [1; 2; 3; 4] son más pequeños que 5:

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

Definir el método `contains` con `exists` :

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

O incluso

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

Ahora verifique si la lista [1 .. 5] contiene el valor 2:

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

Implementación `reverse` :

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

Implementando `map` y `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]
```

¿Hay algo que el `fold` no pueda hacer? No lo sé realmente

Cálculo de la suma de todos los elementos de una lista

Para calcular la suma de términos (de tipo float, int o entero grande) de una lista de números, es preferible usar List.sum. En otros casos, List.fold es la función que mejor se adapta para calcular dicha suma.

1. Suma de numeros complejos

En este ejemplo, declaramos una lista de números complejos y calculamos la suma de todos los términos en la lista.

Al comienzo del programa, agregue una referencia a System.Numerics

abrir System.Numerics

Para calcular la suma, inicializamos el acumulador al número complejo 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
```

Resultado:

```
(3, 51)
```

2. Suma de números de tipo sindical.

Supongamos que una lista se compone de números de tipo union (float o int) y desea calcular la suma de estos números.

Declara antes del siguiente tipo de número:

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

Calcule la suma de números de número de tipo de 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
```

Resultado:

```
13.5
```

El primer parámetro de la función, que representa el acumulador, es de tipo float y el segundo parámetro, que representa un elemento en la lista es de tipo de número. Pero antes de agregar, necesitamos usar una coincidencia de patrón y convertir a tipo float cuando elem es de tipo Int.

Lea Pliegues en línea: https://riptutorial.com/es/fsharp/topic/2250/pliegues

Capítulo 25: Portar C # a F

Examples

POCOs

Algunos de los tipos más simples de clases son POCOs.

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

En F # 3.0, se introdujeron propiedades automáticas similares a propiedades automáticas 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 creación de una instancia de cualquiera es similar,

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

Si puede usar valores inmutables, un tipo de registro es mucho más idiomático F #.

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

Y este registro puede ser creado:

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

Los registros también se pueden crear basándose en otros registros especificando el registro existente y agregando `with` , luego una lista de campos para anular:

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

Clase implementando una interfaz

Las clases implementan una interfaz para cumplir con el contrato de la interfaz. Por ejemplo, una clase C # puede implementar `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();
        }
    }
}
```

Para implementar una interfaz en F #, use la `interface` en la definición de 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 -> ()
```

Lea Portar C # a F # en línea: https://riptutorial.com/es/fsharp/topic/6828/portar-c-sharp-a-f-sharp

Capítulo 26: Procesador de buzones

Observaciones

`MailboxProcessor` mantiene una cola de mensajes interna, donde varios productores pueden publicar mensajes utilizando diversas variantes del método `Post` . Luego, un solo consumidor recupera y procesa estos mensajes (a menos que lo implemente de otra manera) utilizando las variantes de `Retrieve` y `Scan` . Por defecto, tanto producir como consumir los mensajes es seguro para subprocesos.

Por defecto, no hay un manejo de errores proporcionado. Si se lanza una excepción no detectada dentro del cuerpo del procesador, la función del cuerpo finalizará, todos los mensajes en la cola se perderán, no se podrán publicar más mensajes y el canal de respuesta (si está disponible) obtendrá una excepción en lugar de una respuesta. Debe proporcionar todo el manejo de errores en caso de que este comportamiento no se adapte a su caso de uso.

Examples

Hola mundo básico

Primero creamos un simple "¡Hola mundo!" `MailboxProcessor` que procesa un tipo de mensaje e imprime saludos.

Necesitarás el tipo de mensaje. Puede ser cualquier cosa, pero las Uniones Discriminadas son una elección natural aquí, ya que enumeran todos los casos posibles en un solo lugar y puede usar fácilmente la comparación de patrones cuando los procesa.

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

Ahora define el propio procesador. Esto se puede hacer con `MailboxProcessor<'message>.Start` el método estático que devuelve un procesador iniciado listo para hacer su trabajo. También puede usar el constructor, pero luego debe asegurarse de iniciar el procesador más tarde.

```
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 ())
```

El parámetro para `Start` es una función que toma una referencia al propio `MailboxProcessor` (que aún no existe, solo lo está creando, pero estará disponible una vez que se ejecute la función). Eso le da acceso a sus diversos métodos de `Receive` y `Scan` para acceder a los mensajes del buzón. Dentro de esta función, puede hacer cualquier procesamiento que necesite, pero un enfoque habitual es un bucle infinito que lee los mensajes uno por uno y se llama a sí mismo después de cada uno.

Ahora el procesador está listo, ¡pero no sirve para nada! ¿Por qué? Necesitas enviar un mensaje para procesarlo. Esto se hace con las variantes del método de `Post` : usemos la más básica, el de disparar y olvidar.

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

Esto coloca un mensaje en la cola interna del `processor` , el buzón, y se devuelve inmediatamente para que el código de llamada pueda continuar. Una vez que el procesador recupera el mensaje, lo procesará, pero eso se hará de forma asíncrona para publicarlo, y lo más probable es que se realice en un subproceso separado.

Muy pronto verá el mensaje `"Hi, Alice! This is mailbox processor's inner loop!"` impreso en la salida y ya está listo para muestras más complicadas.

Gestión del estado mutable

Los procesadores de buzones se pueden utilizar para administrar el estado mutable de forma transparente y segura para subprocesos. Vamos a construir un contador simple.

```
// 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
```

Ahora vamos a generar algunas operaciones.

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

Y verá el siguiente 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
```

Concurrencia

Dado que el procesador de buzones procesa los mensajes uno por uno y no hay intercalado, también puede producir los mensajes de varios subprocesos y no verá los problemas típicos de las operaciones perdidas o duplicadas. No hay forma de que un mensaje use el estado anterior de otros mensajes, a menos que implemente específicamente el procesador.

```
let processor = createProcessor 0

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

Todos los mensajes son publicados desde diferentes hilos. El orden en el que se publican los mensajes en el buzón no es determinista, por lo que el orden de procesarlos no es determinista, pero como el número total de incrementos y decrementos está equilibrado, verá que el estado final es 0, sin importar el orden y desde qué hilos se enviaron los mensajes.

Verdadero estado mutable

En el ejemplo anterior, solo hemos simulado el estado mutable pasando el parámetro de bucle recursivo, pero el procesador del buzón tiene todas estas propiedades incluso para un estado verdaderamente mutable. Esto es importante cuando se mantiene un estado grande y la inmutabilidad no es práctica por razones de rendimiento.

Podemos reescribir nuestro contador a la siguiente implementación.

```
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 ())
```

Aunque esto definitivamente no sería seguro para subprocesos si el estado del contador se modificara directamente desde varios subprocesos, puede ver mediante el uso de mensajes paralelos en la sección anterior que el procesador del buzón procesa los mensajes uno tras otro sin intercalar, por lo que cada mensaje usa la Valor más actual.

Valores de retorno

Puede devolver de forma asíncrona un valor para cada mensaje procesado si envía un `AsyncReplyChannel<'a>` como parte del mensaje.

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

Luego, el procesador del buzón puede usar este canal cuando procesa el mensaje para enviar un valor al llamante.

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

Ahora para crear un mensaje, necesita el `AsyncReplyChannel<'a>` . ¿Qué es y cómo crea una instancia de trabajo? La mejor manera es dejar que MailboxProcessor se lo proporcione y extraer la respuesta a un `Async<'a>` más común `Async<'a>` . Esto se puede hacer usando, por ejemplo, el método `PostAndAsynReply` , donde no se publica el mensaje completo, sino una función de tipo (en nuestro caso) `AsyncReplyChannel<OutputData> -> MessageWithResponse` :

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

Esto publicará el mensaje en una cola y esperará la respuesta, que llegará una vez que el procesador llegue a este mensaje y responda utilizando el canal.

También hay una variante síncrona `PostAndReply` que bloquea el subproceso de llamada hasta que el procesador responde.

Procesamiento de mensajes fuera de orden

Puede usar los métodos `Scan` o `TryScan` para buscar mensajes específicos en la cola y procesarlos independientemente de cuántos mensajes haya antes de ellos. Ambos métodos miran los mensajes en la cola en el orden en que llegaron y buscarán un mensaje específico (hasta el tiempo de espera opcional). En caso de que no exista tal mensaje, `TryScan` devolverá Ninguno, mientras que la `Scan` seguirá esperando hasta que llegue ese mensaje o la operación se agote.

Vamos a verlo en la práctica. Queremos que el procesador procese las `RegularOperations` cuando sea posible, pero siempre que haya una `PriorityOperation` , debe procesarse lo antes posible, sin importar cuántas `RegularOperations` en la cola.

```
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())
```

Lea Procesador de buzones en línea: https://riptutorial.com/es/fsharp/topic/9409/procesador-de-buzones

Capítulo 27: Proveedores de tipo

Examples

Usando el proveedor de tipos CSV

Dado el siguiente archivo CSV:

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

Puedes leer los datos con el siguiente 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
```

Usando el proveedor de tipos WMI

El proveedor de tipos de WMI le permite consultar los servicios de WMI con una escritura fuerte.

Para generar los resultados de una consulta WMI como 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)
```

Lea Proveedores de tipo en línea: https://riptutorial.com/es/fsharp/topic/1631/proveedores-de-tipo

Capítulo 28: Reflexión

Examples

Reflexión robusta utilizando citas de F

La reflexión es útil pero frágil. Considera esto:

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

Los problemas con este tipo de código son:

1. El código no funciona porque hay varias sobrecargas de `String.StartsWith`
2. Incluso si no hubiera sobrecargas en este momento, las versiones posteriores de la biblioteca podrían agregar una sobrecarga que cause un bloqueo en el tiempo de ejecución
3. Las herramientas de refactorización como los `Rename methods` se rompen con la reflexión.

Esto significa que tenemos un bloqueo en tiempo de ejecución para algo que se conoce como tiempo de compilación. Eso parece subóptimo.

Usando citas de `F#` es posible evitar todos los problemas anteriores. Definimos algunas funciones de ayuda:

```
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
```

Usamos las funciones como esta:

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

Esto imprime:

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

`<@ ... @>` significa que, en lugar de ejecutar la expresión dentro de `F#` genera un árbol de expresión que representa la expresión. `<@ "".StartsWith "" @>` genera un árbol de expresiones que se ve así: `Call (Some (Value ("")), StartsWith, [Value ("")])` . Este árbol de expresiones coincide con lo que espera `getMethodInfo` y devolverá la información del método correcto.

Esto resuelve todos los problemas mencionados anteriormente.

Lea Reflexión en línea: https://riptutorial.com/es/fsharp/topic/4124/reflexion

Capítulo 29: Secuencia

Examples

Generar secuencias

Hay varias formas de crear una secuencia.

Puedes usar funciones del módulo 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
```

También puede utilizar la expresión de secuencia:

```
// 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 }
}
```

Introducción a las secuencias.

Una secuencia es una serie de elementos que pueden ser enumerados. Es un alias de System.Collections.Generic.IEnumerable y perezoso. Almacena una serie de elementos del

mismo tipo (puede ser cualquier valor u objeto, incluso otra secuencia). Las funciones del Seq.module se pueden utilizar para operar en él.

Aquí hay un ejemplo simple de una enumeración de secuencia:

```
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
```

Salida:

```
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; ...]
```

Aplicar una función a cada elemento de una secuencia usando Seq.map

Seq.filter

Supongamos que tenemos una secuencia de enteros y queremos crear una secuencia que contenga solo los enteros pares. Podemos obtener este último utilizando la función de `filter` del módulo Seq. La función de `filter` tiene la firma de tipo `('a -> bool) -> seq<'a> -> seq<'a>` ; esto indica que acepta una función que devuelve verdadero o falso (a veces llamado predicado) para una entrada dada de tipo `'a` y una secuencia que comprende valores de tipo `'a` para producir una secuencia que comprende valores de 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
```

Secuencias repetitivas infinitas

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

Se pueden crear secuencias repetidas usando una expresión de cálculo `seq {}`

Lea Secuencia en línea: https://riptutorial.com/es/fsharp/topic/2354/secuencia

Capítulo 30: Tipos de opciones

Examples

Definición de Opción

Una `Option` es una unión discriminada con dos casos, `None` o `Some` .

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

Utilice la opción <'T> sobre valores nulos

En los lenguajes de programación funcional, como `F#` `null` valores `null` se consideran potencialmente dañinos y de estilo deficiente (no idiomático).

Considere este código `C#` :

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

`x.Length` lanzará si `x` es `null` , agreguemos protección:

```
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;
```

En `null` valores `null` `F#` idiomáticos no se utilizan, por lo que nuestro código se ve así:

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

Sin embargo, a veces es necesario representar valores vacíos o no válidos. Entonces podemos usar la `Option<'T>` :

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

`SomeFunction` devuelve `Some` valor de `string` o `None` . Extraemos el valor de la `string` utilizando el

patrón de coincidencia

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

La razón por la cual este código es menos frágil que:

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

Es porque no podemos llamar `Length` en una `string option` . Necesitamos extraer el valor de la `string` utilizando la coincidencia de patrones y al hacerlo, tenemos la garantía de que el valor de la `string` es seguro de usar.

El módulo opcional habilita la programación orientada al ferrocarril

El manejo de errores es importante pero puede convertir un algoritmo elegante en un desastre. La Programación Orientada a Ferrocarriles (`ROP`) se utiliza para hacer que el manejo de errores sea elegante y componible.

Considere la función simple `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
```

El propósito de `f` es analizar el valor de la `string` entrada (si hay `Some`) en un `int` . Si el `int` es mayor que `0` lanzamos en un `float` . En todos los demás casos, rescatamos con `None` .

Aunque es una función extremadamente simple, la `match` anidada disminuye significativamente la legibilidad.

`ROP` observa que tenemos dos tipos de vías de ejecución en nuestro programa

1. Camino feliz - eventualmente calcularemos `Some` valor
2. Ruta de error: todas las demás rutas producen `None`

Como las rutas de error son más frecuentes, tienden a tomar el control del código. Nos gustaría que el código de ruta feliz sea la ruta de código más visible.

Una función equivalente `g` usa `ROP` podría verse así:

```
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
```

Se parece mucho a cómo tendemos a procesar listas y secuencias en `F#` .

Uno puede ver una `Option<'T>` como una `List<'T>` que solo puede contener `0` o `1` elemento donde `Option.bind` comporta como `List.pick` (conceptualmente `Option.bind` asigna mejor a `List.collect` pero `List.pick` podría ser Más fácil de entender).

`bind` , `filter` y `map` manejan las rutas de error `g` solo contienen el código de ruta feliz.

Todas las funciones que aceptan directamente la `Option<_>` y devuelven la `Option<_>` pueden componer directamente con `|>` y `>>` .

`ROP` por lo tanto aumenta la legibilidad y la composibilidad.

Usando tipos de opciones de C

No es una buena idea exponer los tipos de opción al código C #, ya que C # no tiene una forma de manejarlos. Las opciones son introducir `FSharp.Core` como una dependencia en su proyecto de C # (que es lo que tendría que hacer si está consumiendo una biblioteca de F # *no* diseñada para interoperar con C #), o cambiar los valores de `None` a `null` .

Pre-F # 4.0

La forma de hacerlo es crear una función de conversión propia:

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

Para los tipos de valor tendría que recurrir a boxearlos o usar `System.Nullable` .

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

F # 4.0

En F # 4.0, las funciones `ofObj` , `toObj` , `ofNullable` y `toNullable` introdujeron en el módulo de `Option` . En F # interactivo se pueden utilizar de la siguiente manera:

```
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
```

Tenga en cuenta que `None` compila a `null` internamente. Sin embargo, en lo que respecta a F #, es 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
```

Lea Tipos de opciones en línea: https://riptutorial.com/es/fsharp/topic/3175/tipos-de-opciones

Capítulo 31: Unidades de medida

Observaciones

Unidades en tiempo de ejecución

Las unidades de medida se usan solo para la verificación estática por parte del compilador, y no están disponibles en tiempo de ejecución. No se pueden utilizar en la reflexión o en métodos como `ToString` .

Por ejemplo, C # da un `double` sin unidades para un campo de tipo `float<m>` definido y expuesto desde una biblioteca F #.

Examples

Asegurando Unidades Consistentes en Cálculos

Las unidades de medida son anotaciones de tipo adicionales que se pueden agregar a flotantes o enteros. Se pueden usar para verificar, en el momento de la compilación, que los cálculos utilizan unidades de manera consistente.

Para definir anotaciones:

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

Una vez definidas, las anotaciones se pueden usar para verificar que una expresión dé como resultado el tipo esperado.

```
// 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
```

Conversiones entre unidades

```
[<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>
```

Tenga en cuenta que el compilador F # no sabe que `1<m>` es igual a `100<cm>` . En cuanto a lo que importa, las unidades son tipos separados. Puede escribir funciones similares para convertir de metros a kilogramos, y al compilador no le importará.

```
[<Measure>] type kg

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

No es posible definir unidades de medida como múltiplos de otras unidades como

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

Sin embargo, definir unidades "por algo", por ejemplo Hertz, medir la frecuencia, es simplemente "por segundo", es bastante simple.

```
// 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
```

Usando LanguagePrimitives para preservar o establecer unidades

Cuando una función no conserva las unidades automáticamente debido a las operaciones de nivel inferior, el módulo `LanguagePrimitives` se puede usar para establecer unidades en las primitivas que las soportan:

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

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

Para asignar unidades de medida a un valor de punto flotante de precisión doble, simplemente multiplique por uno con las unidades correctas:

```
[<Measure>]
type USD

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

Para asignar unidades de medida a un valor sin unidades que no sea Sistema. Doble, por ejemplo, al llegar de una biblioteca escrita en otro idioma, use una conversión:

```
open LanguagePrimitives

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

Aquí están los tipos de funciones reportadas por F # interactivo:

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

Parámetros del tipo de unidad de medida

El atributo `[<Measure>]` se puede usar en los parámetros de tipo para declarar tipos que son genéricos con respecto a las unidades de medida:

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

Uso de prueba:

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

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

Usa tipos de unidades estandarizadas para mantener la compatibilidad

Por ejemplo, los tipos para las unidades SI se han estandarizado en la biblioteca central F #, en `Microsoft.FSharp.Data.UnitSystems.SI` . Abra el `UnitNames` apropiado, `UnitNames` o `UnitSymbols` , para usarlos. O, si solo se requieren unas pocas unidades SI, se pueden importar con alias de tipo:

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

Algunos usuarios tienden a hacer lo siguiente, que **no debe hacerse** cuando ya hay una definición disponible:

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

La diferencia se hace evidente al interactuar con otro código que se refiere a los tipos de SI estándar. El código que hace referencia a las unidades estándar es compatible, mientras que el código que define su propio tipo es incompatible con cualquier código que no utilice su definición específica.

Por lo tanto, siempre use los tipos estándar para las unidades SI. No importa si hace referencia a `UnitNames` o `UnitSymbols` , ya que los nombres equivalentes dentro de esos dos se refieren al mismo 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>
```

Lea Unidades de medida en línea: https://riptutorial.com/es/fsharp/topic/1055/unidades-de-medida

Capítulo 32: Uniones discriminadas

Examples

Nombrando elementos de tuplas dentro de uniones discriminadas

Al definir uniones discriminadas, puede nombrar elementos de tipos de tuplas y usar estos nombres durante la comparación de patrones.

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

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

Además, nombrar los elementos de uniones discriminadas mejora la legibilidad del código y la interoperabilidad con C #; los nombres proporcionados se utilizarán para los nombres de las propiedades y los parámetros de los constructores. Los nombres generados predeterminados en el código de interoperabilidad son "Elemento", "Elemento1", "Elemento2" ...

Uso Básico Discriminado de la Unión

Las uniones discriminadas en F # ofrecen una forma de definir tipos que pueden contener cualquier número de tipos de datos diferentes. Su funcionalidad es similar a las uniones C ++ o variantes de VB, pero con el beneficio adicional de ser de tipo seguro.

```
// 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
```

Uniones al estilo Enum

La información de tipo no necesita ser incluida en los casos de una unión discriminada. Al omitir la información de tipo, puede crear una unión que simplemente representa un conjunto de opciones,

similar a una enumeración.

```
// 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
```

Convertir hacia y desde cuerdas con Reflexión

A veces es necesario convertir una Unión Discriminada hacia y desde una cadena:

```
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
```

Unión individual discriminada caso

Una unión discriminada de un solo caso es como cualquier otra unión discriminada, excepto que solo tiene un 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
```

Es útil para hacer cumplir la seguridad de tipos y se usa comúnmente en F # en lugar de C # y Java, donde la creación de nuevos tipos conlleva más sobrecarga.

Las siguientes dos definiciones de tipos alternativos dan como resultado que se declare la misma unión discriminada de un solo caso:

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

Uso de uniones discriminadas de un solo caso como registros

A veces es útil crear tipos de unión con un solo caso para implementar tipos similares a registros:

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)

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

Se vuelven muy útiles porque pueden descomponerse mediante la coincidencia de patrones de la misma manera que los argumentos de la tupla pueden:

```
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 el atributo `RequireQualifiedAccess` , los casos de unión deben denominarse `MyUnion.MyCase` lugar de solo `MyCase` . Esto evita las colisiones de nombres en el espacio de nombres o el módulo adjunto:

```
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
```

Si, por ejemplo, el `System` se ha abierto, `Single` refiere a `System.Single` . No hay colisión con los `Requirements.Single` caso del sindicato. Solo.

Uniones recursivas discriminadas

Tipo recursivo

Las uniones discriminadas pueden ser recursivas, es decir, pueden referirse a sí mismas en su definición. El primer ejemplo aquí es un árbol:

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

Como ejemplo, definamos el siguiente árbol:

```
    1
  2   5
3   4
```

Podemos definir este árbol usando nuestra unión recursiva discriminada de la siguiente manera:

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

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

Iterar sobre el árbol es solo una cuestión de coincidencia de patrones:

```
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]
```

Tipos recursivos mutuamente dependientes

Una forma de lograr la recursión es tener tipos anidados mutuamente dependientes.

```
// 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 definición de un tipo de registro directamente dentro de una unión discriminada está en desuso:

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

Puede usar la palabra clave `and` para encadenar definiciones mutuamente dependientes:

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

Lea Uniones discriminadas en línea: https://riptutorial.com/es/fsharp/topic/1025/uniones-discriminadas

Capítulo 33: Uso de F #, WPF, FsXaml, un menú y un cuadro de diálogo

Introducción

El objetivo aquí es crear una aplicación simple en F # utilizando Windows Presentation Foundation (WPF) con menús y cuadros de diálogo tradicionales. Proviene de mi frustración al tratar de leer cientos de secciones de documentación, artículos y publicaciones relacionadas con F # y WPF. Para hacer cualquier cosa con WPF, parece que tienes que saberlo todo al respecto. Mi propósito aquí es proporcionar una forma posible de, un proyecto de escritorio simple que pueda servir como una plantilla para sus aplicaciones.

Examples

Configurar el proyecto

Asumiremos que está haciendo esto en Visual Studio 2015 (VS 2015 Community, en mi caso). Crear un proyecto de consola vacío en VS. En proyecto | Las propiedades cambian el tipo de salida a la aplicación de Windows.

A continuación, use NuGet para agregar FsXaml.Wpf al proyecto; este paquete fue creado por la estimable Reed Copsey, Jr., y simplifica enormemente el uso de WPF desde F #. En la instalación, agregará una serie de otros conjuntos de WPF, por lo que no tendrá que hacerlo. Hay otros paquetes similares a FsXaml, pero uno de mis objetivos era mantener el número de herramientas lo más pequeño posible para que el proyecto en general sea lo más sencillo y fácil de mantener posible.

Además, agregue UIAutomationTypes como referencia; viene como parte de .NET.

Añadir la "lógica de negocios"

Presumiblemente, su programa hará algo. Agregue su código de trabajo al proyecto en lugar de Program.fs. En este caso, nuestra tarea es dibujar curvas de espirógrafo en un lienzo de ventana. Esto se logra utilizando Spirograph.fs, a continuación.

```
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 es el primer archivo F # en el orden de compilación, por lo que contiene las definiciones de los tipos que necesitaremos. Su trabajo es dibujar un espirógrafo en el lienzo de la ventana principal según los parámetros ingresados en un cuadro de diálogo. Ya que hay muchas referencias sobre cómo dibujar un espirógrafo, no entraremos en eso aquí.

Crea la ventana principal en XAML.

Debe crear un archivo XAML que defina la ventana principal que contiene nuestro menú y espacio de dibujo. Aquí está el código XAML en 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>
```

Los comentarios generalmente no se incluyen en los archivos XAML, lo que creo que es un error. He añadido algunos comentarios a todos los archivos XAML en este proyecto. No afirmo que sean los mejores comentarios que se hayan escrito, pero al menos muestran cómo se debe formatear un comentario. Tenga en cuenta que los comentarios anidados no están permitidos en XAML.

Crear el cuadro de diálogo en XAML y F

El archivo XAML para los parámetros del espirógrafo se encuentra a continuación. Incluye tres cuadros de texto para los parámetros del espirógrafo y un grupo de tres botones de radio para el color. Cuando le damos a los botones de radio el mismo nombre de grupo, como lo tenemos aquí, WPF maneja el encendido / apagado cuando se selecciona 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>
```

Ahora agregamos el código detrás de Dialog.Box. Por convención, el código utilizado para manejar la interfaz del cuadro de diálogo con el resto del programa se llama XXX.xaml.fs, donde el archivo XAML asociado se llama 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 código aquí está dedicado a garantizar que los parámetros del espirógrafo en Spirograph.fs coincidan con los que se muestran en este cuadro de diálogo. Tenga en cuenta que no hay comprobación de errores: si ingresa un punto flotante para los enteros esperados en los dos campos de parámetros superiores, el programa se bloqueará. Entonces, por favor agregue la comprobación de errores en su propio esfuerzo.

Tenga en cuenta también que los campos de entrada de parámetros están deshabilitados con un color aleatorio seleccionado en los botones de opción. Está aquí solo para mostrar cómo se puede hacer.

Con el fin de mover los datos entre el cuadro de diálogo y el programa, uso System.Windows.Element.FindName () para encontrar el control adecuado, enviarlo al control que debe ser y luego obtener la configuración relevante de la Controlar. La mayoría de los otros programas de ejemplo utilizan enlace de datos. No lo hice por dos razones: primero, no pude averiguar cómo hacerlo funcionar, y segundo, cuando no funcionó no recibí ningún mensaje de error de ningún tipo. Tal vez alguien que visite esto en StackOverflow pueda decirme cómo usar el enlace de datos sin incluir un nuevo conjunto de paquetes NuGet.

Agregue el código detrás de 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
```

No hay mucho que hacer aquí: abrimos el cuadro de diálogo Parámetros cuando es necesario y tenemos la opción de volver a dibujar el espirógrafo con los parámetros actuales.

Agregue App.xaml y App.xaml.fs para unir todo

```
<!-- 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>
```

Aquí está el código detrás:

```
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 está completo aquí, principalmente para mostrar dónde se pueden declarar los recursos de la aplicación, como iconos, gráficos o archivos externos. App.xaml.fs complementario reúne el modelo y la ventana principal, ajusta el tamaño de la ventana principal a dos tercios del tamaño de la pantalla disponible y lo ejecuta.

Cuando compile esto, recuerde asegurarse de que la propiedad Build para cada archivo xaml esté establecida en Resource. Luego puede ejecutar el depurador o compilarlo en un archivo exe. Tenga en cuenta que no puede ejecutar esto utilizando el intérprete de F #: el paquete FsXaml y el intérprete son incompatibles.

Ahí tienes. Espero que pueda usar esto como punto de partida para sus propias aplicaciones, y al hacerlo, puede ampliar su conocimiento más allá de lo que se muestra aquí. Cualquier comentario y sugerencias serán apreciados.

Lea Uso de F #, WPF, FsXaml, un menú y un cuadro de diálogo en línea: https://riptutorial.com/es/fsharp/topic/9145/uso-de-f-sharp--wpf--fsxaml--un-menu-y-un-cuadro-de-dialogo

Creditos

S. No	Capítulos	Contributors
1	Empezando con F #	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	1: Código F # WPF detrás de la aplicación con FsXaml	Bent Tranberg
3	Archivos	eirik, goric, Ringil
4	El tipo de "unidad"	4444, Abel, Jake Lishman, rmunn
5	Evaluación perezosa	inzi
6	Extensiones de Tipo y Módulo	Jono Job
7	F # Consejos y trucos de rendimiento	FuleSnabel, Paul Westcott, Ringil, s952163
8	F # en .NET Core	Boggin, Joel Martinez
9	Flujos de trabajo de secuencia	Jwosty
10	Funciones	asibahi, Julien Pires, rmunn, ronilk
11	Genéricos	Jake Lishman
12	Implementación de patrones de diseño en F #	FuleSnabel, Ringil
13	Instrumentos de cuerda	FireAlkazar, Julien Pires
14	Introducción a WPF en F #	Funk
15	La coincidencia de patrones	asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene

16	Las clases	asibahi, inzi, RamenChef, Tomasz Maczyński
17	Liza	asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj
18	Los operadores	FuleSnabel
19	Los tipos	Cedric Royer-Bertrand, FuleSnabel, Julien Pires
20	Memorización	Jean-Claude Colette, Julien Pires, Ringil
21	Mónadas	FuleSnabel
22	Parámetros de tipo resueltos estáticamente	Maslow
23	Patrones activos	Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
24	Pliegues	Jean-Claude Colette, Zaid Ajaj
25	Portar C # a F #	jdphenix, marklam, RamenChef
26	Procesador de buzones	Honza Brestan
27	Proveedores de tipo	GregC, jdphenix, Joel Martinez
28	Reflexión	FuleSnabel
29	Secuencia	Foggy Finder, inzi, James McCalden, Julien Pires, s952163
30	Tipos de opciones	asibahi, chillitom, FuleSnabel
31	Unidades de medida	asibahi, goric, GregC, Vandroiy
32	Uniones discriminadas	chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen, Vandroiy, VillasV
33	Uso de F #, WPF, FsXaml, un menú y un cuadro de diálogo	Bob McCrory, Goswin