eBook Gratuit

APPRENEZ F#

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#f#

Table des matières

À propos

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

Chapitre 1: Commencer avec F

Remarques

F # est une langue "fonctionnelle-première". Vous pouvez en apprendre davantage sur tous les différents types d'expressions , ainsi que des fonctions .

Le compilateur F # - qui est open source - compile vos programmes en IL, ce qui signifie que vous pouvez utiliser le code F # à partir de tout langage compatible .NET tel que C # ; et l'exécuter sur Mono, .NET Core ou .NET Framework sous Windows.

Versions

Version	Date de sortie
1 fois	2005-05-01
2.0	2010-04-01
3.0	2012-08-01
3.1	2013-10-01
4.0	2015-07-01

Examples

Installation ou configuration

les fenêtres

Si Visual Studio (toute version incluant Express et Community) est installé, F # devrait déjà être inclus. Choisissez simplement F # comme langue lorsque vous créez un nouveau projet. Ou consultez http://fsharp.org/use/windows/ pour plus d'options.

OS X

Xamarin Studio prend en charge F #. Vous pouvez également utiliser VS Code pour OS X , un éditeur multi-plateforme de Microsoft.
Une fois l'installation terminée, lancez `VS Code Quick Open` (`Ctrl` + `P`), puis exécutez `ext install Ionide-fsharp`

Vous pouvez également envisager Visual Studio pour Mac .

Il existe d'autres alternatives décrites ici .

Linux

Installez les packages `mono-complete` et `fsharp` via le gestionnaire de paquets de votre distribution (Apt, Yum, etc.). Pour une bonne expérience de montage, utilisez Visual Studio Code et installez le `ionide-fsharp` in `ionide-fsharp` ou utilisez Atom et installez le `ionide-installer` in `ionide-installer` . Voir http://fsharp.org/use/linux/ pour plus d'options.

Bonjour le monde!

Ceci est le code pour un projet de console simple, qui affiche "Hello, World!" à STDOUT, et quitte avec un code de sortie de `0`

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

Exemple de décomposition Ligne par ligne:

- `[<EntryPoint>]` - Un attribut .net qui marque "la méthode que vous utilisez pour définir le point d'entrée" de votre programme (source).
- `let main argv` - définit une fonction appelée `main` avec un seul paramètre `argv` . Comme il s'agit du point d'entrée du programme, `argv` sera un tableau de chaînes. Le contenu du tableau sont les arguments transmis au programme lors de son exécution.
- `printfn "Hello, World!"` - la fonction `printfn` la chaîne ** passée en premier argument, en ajoutant également une nouvelle ligne.
- `0` - F # renvoient toujours une valeur et la valeur renvoyée est le résultat de la dernière expression de la fonction. Mettre `0` comme dernière ligne signifie que la fonction retournera toujours zéro (un entier).

** Ce n'est en fait *pas* une chaîne même si elle en a l'air. C'est en fait un TextWriterFormat , qui permet éventuellement d'utiliser des arguments vérifiés de type statique. Mais dans le cas d'un exemple de "bonjour le monde", on peut le considérer comme une chaîne.

F # Interactive

F # Interactive, est un environnement REPL qui vous permet d'exécuter du code F #, une ligne à la fois.

Si vous avez installé Visual Studio avec F #, vous pouvez exécuter F # Interactive dans la console en tapant `"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"` . Sous Linux ou OS X, la commande est `fsharpi` à la place, qui devrait être soit dans `/usr/bin` ou `/usr/local/bin` selon la façon dont vous avez installé F # - De toute façon, la commande doit être sur votre `PATH`

afin que vous puissiez tapez simplement `fsharpi` .

Exemple d'utilisation interactive 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
```

Utilisez `#help;;` pour aider

S'il vous plaît noter l'utilisation de `;;` pour indiquer à REPL d'exécuter les commandes précédemment saisies.

Lire Commencer avec F # en ligne: https://riptutorial.com/fr/fsharp/topic/817/commencer-avec-f-sharp

Chapitre 2: 1: F # Code WPF derrière l'application avec FsXaml

Introduction

La plupart des exemples trouvés pour la programmation F # WPF semblent concerner le pattern MVVM, et quelques-uns avec MVC, mais il n'y en a aucun qui montre correctement comment se lancer avec du "bon vieux" code derrière.

Le code derrière le modèle est très facile à utiliser pour l'enseignement ainsi que pour l'expérimentation. Il est utilisé dans de nombreux livres d'introduction et matériel pédagogique sur le Web. C'est pourquoi.

Ces exemples montreront comment créer un code derrière une application avec des fenêtres, des contrôles, des images et des icônes, etc.

Examples

Créez un nouveau code F # WPF derrière l'application.

Créez une application console F #.

Modifiez le **type** de **sortie** de l'application en application *Windows* .

Ajoutez le package **FsXaml** NuGet.

Ajoutez ces quatre fichiers source, dans l'ordre indiqué ici.

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.
```

Supprimez le fichier *Program.fs* du projet.

Modifiez l' **action** de génération en *ressource* pour les deux fichiers xaml.

Ajoutez une référence à l'assembly .NET **UIAutomationTypes** .

Compiler et exécuter

Vous ne pouvez pas utiliser le concepteur pour ajouter des gestionnaires d'événements, mais ce n'est pas un problème du tout. Ajoutez-les simplement manuellement dans le code derrière, comme vous le voyez avec les trois gestionnaires de cet exemple, y compris le gestionnaire du bouton de test.

UPDATE: Une manière alternative et probablement plus élégante d'ajouter des gestionnaires d'événements a été ajoutée à FsXaml. Vous pouvez ajouter le gestionnaire d'événements dans XAML, comme dans C #, mais vous devez le faire manuellement, puis remplacer le membre correspondant qui apparaît dans votre type F #. Je recommande ceci

3: Ajouter une icône à une fenêtre

C'est une bonne idée de conserver toutes les icônes et images dans un ou plusieurs dossiers.

Cliquez avec le bouton droit sur le projet et utilisez F # Power Tools / New Folder pour créer un dossier nommé Images.

Sur le disque, placez votre icône dans le nouveau dossier *Images* .

De retour dans Visual Studio, cliquez avec le bouton droit sur *Images* et utilisez **Ajouter / Élément existant** , puis affichez *Tous les fichiers (.*) ** pour afficher le fichier d'icône afin de pouvoir le sélectionner, puis **ajoutez-** le.

Sélectionnez le fichier d'icône et définissez son **action** de *génération* sur *ressource* .

Dans MainWindow.xaml, utilisez l'attribut Icon comme ceci. Les lignes environnantes sont affichées pour le contexte.

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

Avant de vous lancer, effectuez une reconstruction, et pas seulement une génération. En effet, Visual Studio ne met pas toujours le fichier d'icône dans l'exécutable, sauf si vous le reconstruisez.

C'est la fenêtre, et non l'application, qui a maintenant une icône. Vous verrez l'icône en haut à gauche de la fenêtre à l'exécution et vous la verrez dans la barre des tâches. Le Gestionnaire des tâches et l'Explorateur de fichiers Windows n'afficheront pas cette icône, car ils affichent l'icône de l'application plutôt que l'icône de la fenêtre.

4: Ajouter une icône à l'application

Créez un fichier texte nommé AppIcon.rc avec le contenu suivant.

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

Vous aurez besoin d'un fichier icône nommé AppIcon.ico pour que cela fonctionne, mais vous pouvez bien sûr ajuster les noms à votre convenance.

Exécutez la commande suivante.

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

Si vous ne trouvez pas rc.exe à cet emplacement, recherchez-le sous **C: \ Program Files (x86) \ Windows Kits** . Si vous ne le trouvez toujours pas, téléchargez Windows SDK de Microsoft.

Un fichier nommé AppIcon.res sera généré.

Dans Visual Studio, ouvrez les propriétés du projet. Sélectionnez la page de l' **application** .

Dans la zone de texte intitulée **Fichier de ressources** , tapez *AppIcon.res* (ou *Images* \ *AppIcon.res* si vous le placez), puis fermez les propriétés du projet à enregistrer.

Un message d'erreur apparaîtra, indiquant "Le fichier de ressources entré n'existe pas. Ignorez-le. Le message d'erreur ne réapparaîtra pas.

Reconstruire. L'exécutable aura alors une icône d'application, et cela apparaît dans l'Explorateur de fichiers. En cours d'exécution, cette icône apparaîtra également dans le Gestionnaire des tâches.

2: Ajouter un contrôle

Ajoutez ces deux fichiers dans cet ordre au-dessus des fichiers de la fenêtre principale.

MyControl.xaml

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

MyControl.xaml.fs

```
namespace FirstDemo

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

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

L' **action** de *génération* pour *MyControl.xaml* doit être définie sur *Resource* .

Vous devrez bien sûr ajouter plus tard "as this" dans la déclaration de MyControl, comme pour la fenêtre principale.

Dans le fichier **MainWindow.xaml.fs** , dans la classe pour MainWindow, ajoutez cette ligne

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

et ajoutez ces deux lignes dans la **do** -section de la classe de la fenêtre principale.

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

Il peut y avoir plus d'une **do-section** dans une classe et vous en aurez probablement besoin lorsque vous écrirez beaucoup de code code-behind.

Une couleur de fond vert clair a été attribuée au contrôle pour que vous puissiez facilement voir où il se trouve.

Sachez que le contrôle bloquera le bouton de la fenêtre principale de la vue. Il est hors de la portée de ces exemples de vous apprendre WPF en général, nous ne pourrons donc pas résoudre ce problème ici.

Comment ajouter des contrôles à partir de bibliothèques tierces

Si vous ajoutez des contrôles de bibliothèques tierces dans un projet C # WPF, le fichier XAML aura normalement des lignes comme celle-ci.

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

Cela ne fonctionnera peut-être pas avec FsXaml.

Le concepteur et le compilateur acceptent cette ligne, mais il y aura probablement une exception au moment de l'exécution pour se plaindre du fait que le type tiers n'est pas trouvé lors de la lecture du fichier XAML.

Essayez plutôt quelque chose comme ce qui suit.

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

Ceci est un exemple de contrôle qui dépend de ce qui précède.

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

La bibliothèque utilisée dans cet exemple est Extended Toolkit Wpf, disponible gratuitement via NuGet ou en tant qu'installateur. Si vous téléchargez des bibliothèques via NuGet, les contrôles ne sont pas disponibles dans la boîte à outils, mais ils s'affichent toujours dans le concepteur si vous les ajoutez manuellement dans XAML et que les propriétés sont disponibles dans le volet Propriétés.

Lire 1: F # Code WPF derrière l'application avec FsXaml en ligne:
https://riptutorial.com/fr/fsharp/topic/9008/1--f-sharp-code-wpf-derriere-l-application-avec-fsxaml

Chapitre 3: Cordes

Examples

Littéraux de chaîne

```
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
```

Formatage de chaîne simple

Il y a plusieurs façons de formater et d'obtenir une chaîne en conséquence.

La manière .NET consiste à utiliser `String.Format` ou `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 # a également des fonctions pour formater la chaîne dans un style C. Il y a des équivalents pour chaque fonction .NET:

- `sprintf` (String.Format):

```
open System

let hello = sprintf "Hello %s" "World"
// "Hello World", "%s" is for string

let helloInt = sprintf "Hello %i" 42
```

```
// "Hello 42", "%i" is for int

let helloFloat = sprintf "Hello %f" 4.2
// "Hello 4.2000", "%f" is for float

let helloBool = sprintf "Hello %b" true
// "Hello true", "%b" is for bool

let helloNativeType = sprintf "Hello %A again!" ("World", DateTime.Now)
// "Hello {formatted date}", "%A" is for native type

let helloObject = sprintf "Hello %O again!" DateTime.Now
// "Hello {formatted date}", "%O" is for calling ToString
```

- `bprintf` (StringBuilder.AppendFormat):

```
open System
open System.Text

let builder = StringBuilder()

// Attach the StringBuilder to the format function with partial application
let append format = Printf.bprintf builder format

// Same behavior as sprintf but strings are appended to a StringBuilder
append "Hello %s again!\n" "World"
append "Hello %i again!\n" 42
append "Hello %f again!\n" 4.2
append "Hello %b again!\n" true
append "Hello %A again!\n" ("World", DateTime.Now)
append "Hello %O again!\n" DateTime.Now

builder.ToString() // Get the result string
```

L'utilisation de ces fonctions au lieu des fonctions .NET offre certains avantages:

- Type de sécurité
- Application partielle
- Prise en charge de type natif F #

Lire Cordes en ligne: https://riptutorial.com/fr/fsharp/topic/1397/cordes

Chapitre 4: Correspondance de motif

Remarques

La correspondance de motifs est une fonctionnalité puissante de nombreux langages fonctionnels, car elle permet souvent de gérer les branchements de manière très succincte par rapport à l'utilisation de plusieurs instructions de style `if` / `else if` / `else` . Cependant, compte tenu des options suffisantes et des gardes «lorsque» , la correspondance des motifs peut aussi devenir verbeuse et difficile à comprendre en un coup d'œil.

Lorsque cela se produit, les patterns actifs de F # peuvent être un excellent moyen de donner des noms significatifs à la logique correspondante, ce qui simplifie le code et permet également la réutilisation.

Examples

Options de correspondance

La correspondance de modèle peut être utile pour gérer les options:

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

La correspondance de modèle vérifie que tout le domaine est couvert

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

donne un avertissement

> C: \ Program Files (x86) \ Code Microsoft Microsoft \ Untitled-1 (2,7): avertissement FS0025: correspond à un motif incomplet sur cette expression. Par exemple, la valeur 'false' peut indiquer un cas non couvert par le ou les motifs.

C'est parce que toutes les valeurs bool possibles n'ont pas été couvertes.

les bools peuvent être explicitement listés mais ints sont plus difficiles à lister

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

Ici, nous utilisons le caractère spécial _ . Le _ correspond à tous les autres cas possibles.

Le _ peut vous causer des ennuis

considérer un type que nous créons nous-mêmes, il ressemble à ceci

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

Nous pourrions écrire un match avec expession qui ressemble à ceci

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

Le code ci-dessus a du sens. Nous supposons que si vous n'êtes pas sobre, vous devriez appeler un uber afin que nous utilisions le _ pour indiquer que

Nous réorganisons plus tard notre code à ceci

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

Le compilateur F # devrait nous avertir et nous demander de modifier notre expression de correspondance pour que la personne recherche des soins médicaux. Au lieu de cela, l'expression du match traite silencieusement la personne inconsciente comme si elle était seulement éméché. Le fait est que vous devez choisir de lister explicitement les cas si possible pour éviter les erreurs de logique.

Les cas sont évalués de haut en bas et la première correspondance est utilisée

Utilisation incorrecte:

Dans l'extrait suivant, la dernière correspondance ne sera jamais utilisée:

```
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"
```

estampes

> x est tout ce qui n'a pas été listé ci-dessus

Usage correct:

Ici, `x = 1` et `x = 4` atteindront leurs cas spécifiques, alors que tout le reste tombera dans le cas par défaut _ :

```
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"
```

estampes

> x est 4

Lorsque les gardes vous permettent d'ajouter des conditions arbitraires

```
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"
```

Lire Correspondance de motif en ligne: https://riptutorial.com/fr/fsharp/topic/1335/correspondance-de-motif

Chapitre 5: Des classes

Examples

Déclarer une classe

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

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

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

Créer une instance

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

Lire Des classes en ligne: https://riptutorial.com/fr/fsharp/topic/3003/des-classes

Chapitre 6: Des dossiers

Examples

Ajouter des fonctions membres aux enregistrements

```
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
```

Utilisation de base

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

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

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

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

Lire Des dossiers en ligne: https://riptutorial.com/fr/fsharp/topic/1136/des-dossiers

Chapitre 7: Des listes

Syntaxe

- [] // une liste vide.

  head :: tail // une cellule de construction contenant un élément, une tête et une liste, tail. :: s'appelle l'opérateur Cons.

  let list1 = [1; 2; 3] // Notez l'utilisation d'un point-virgule.

  let list2 = 0 :: list1 // le résultat est [0; 1; 2; 3]

  let list3 = list1 @ list2 // le résultat est [1; 2; 3; 0; 1; 2; 3]. @ est l'opérateur append.

  let list4 = [1..3] // le résultat est [1; 2; 3]

  let list5 = [1..2..10] // le résultat est [1; 3; 5; 7; 9]

  let list6 = [pour i dans 1..10 faire si i% 2 = 1 alors céder i] // le résultat est [1; 3; 5; 7; 9]

Examples

Utilisation de liste de base

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

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

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

patternMatch list1

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

Calcul de la somme totale des nombres dans une liste

Par récursion

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

L'exemple ci-dessus dit: "Regardez la `list` , est-il vide? Retournez 0. Sinon, c'est une liste non vide. Donc, cela pourrait être `[1]` , `[1; 2]` , [1; 2; 3] etc. Si `list` est [1] alors liez la variable `head` à `1` et `tail` à `[]` puis exécutez `head + sumTotal tail` .

Exemple d'exécution:

```
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
```

Une manière plus générale d'encapsuler le modèle ci-dessus consiste à utiliser des plis fonctionnels! `sumTotal` devient ceci:

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

Créer des listes

Une façon de créer une liste consiste à placer des éléments entre deux crochets, séparés par des points-virgules. Les éléments doivent avoir le même type.

Exemple:

```
> 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]
```

Lorsqu'une liste n'a pas d'élément, elle est vide. Une liste vide peut être déclarée comme suit:

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

Autre exemple

Pour créer une liste d'octets, il suffit de lancer les entiers:

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

Il est également possible de définir des listes de fonctions, d'éléments d'un type défini précédemment, d'objets d'une classe, etc.

Exemple

```
> 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]
```

Gammes

Pour certains types d'éléments (int, float, char, ...), il est possible de définir une liste à l'aide de l'élément start et de l'élément end, en utilisant le modèle suivant:

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

Exemples:

```
> 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]
```

Vous pouvez également spécifier une étape pour certains types, avec le modèle suivant:

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

Exemples:

```
> 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]
```

Générateur

Une autre façon de créer une liste est de la générer automatiquement en utilisant un générateur.

Nous pouvons utiliser l'un des modèles suivants:

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

ou

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

Exemples

```
> 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]
```

Les opérateurs

Certains opérateurs peuvent être utilisés pour construire des listes:

Opérateur Cons ::

Cet opérateur :: est utilisé pour ajouter un élément head à une liste:

```
> 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]
```

Enchaînement

La concaténation des listes est effectuée avec l'opérateur @.

```
> 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]
```

Lire Des listes en ligne: https://riptutorial.com/fr/fsharp/topic/1268/des-listes

Chapitre 8: En utilisant F #, WPF, FsXaml, un menu et une boîte de dialogue

Introduction

L'objectif ici est de créer une application simple en F # à l'aide de Windows Presentation Foundation (WPF) avec des menus et des boîtes de dialogue traditionnels. Cela découle de ma frustration en essayant de parcourir des centaines de sections de documentation, d'articles et de publications traitant de F # et de WPF. Pour faire quelque chose avec WPF, vous semblez devoir tout savoir. Mon but ici est de fournir un moyen possible, un simple projet de bureau pouvant servir de modèle pour vos applications.

Examples

Mettre en place le projet

Nous supposerons que vous faites cela dans Visual Studio 2015 (Communauté VS 2015, dans mon cas). Créez un projet de console vide dans VS. Dans le projet | Les propriétés changent le type de sortie en application Windows.

Ensuite, utilisez NuGet pour ajouter FsXaml.Wpf au projet; Ce paquet a été créé par l'estimable Reed Copsey, Jr., et il simplifie grandement l'utilisation de WPF à partir de F #. Lors de l'installation, il ajoutera un certain nombre d'autres assemblys WPF, vous n'aurez donc pas à le faire. Il existe d'autres logiciels similaires à FsXaml, mais l'un de mes objectifs était de réduire au maximum le nombre d'outils afin de rendre le projet aussi simple et viable que possible.

De plus, ajoutez UIAutomationTypes comme référence; il fait partie de .NET.

Ajouter le "Business Logic"

Votre programme fera probablement quelque chose. Ajoutez votre code de travail au projet à la place de Program.fs. Dans ce cas, notre tâche consiste à dessiner des courbes de spirographe sur un canevas de fenêtre. Ceci est accompli en utilisant Spirograph.fs, ci-dessous.

```
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 est le premier fichier F # dans l'ordre de compilation, il contient donc les définitions des types dont nous aurons besoin. Son travail consiste à dessiner un spirographe dans la fenêtre principale Canvas en fonction des paramètres saisis dans une boîte de dialogue. Comme il y a beaucoup de références sur la façon de dessiner un spirographe, nous n'entrerons pas dans les détails ici.

Créer la fenêtre principale dans XAML

Vous devez créer un fichier XAML qui définit la fenêtre principale contenant notre menu et l'espace de dessin. Voici le code XAML dans 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>
```

Les commentaires ne sont généralement pas inclus dans les fichiers XAML, ce qui est une erreur, à mon avis. J'ai ajouté des commentaires à tous les fichiers XAML de ce projet. Je ne prétends pas que ce sont les meilleurs commentaires jamais écrits, mais ils montrent au moins comment un commentaire doit être formaté. Notez que les commentaires imbriqués ne sont pas autorisés dans XAML.

Créez la boîte de dialogue dans XAML et F

Le fichier XAML pour les paramètres du spirographe est ci-dessous. Il comprend trois zones de texte pour les paramètres du spirographe et un groupe de trois boutons radio pour la couleur. Lorsque nous donnons aux boutons radio le même nom de groupe - comme ici - WPF gère le basculement quand on est sélectionné.

```
<!-- 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>
```

Maintenant, nous ajoutons le code derrière le Dialog.Box. Par convention, le code utilisé pour gérer l'interface de la boîte de dialogue avec le reste du programme s'appelle XXX.xaml.fs, où le fichier XAML associé s'appelle 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
    ()
```

Une grande partie du code ici est consacrée à s'assurer que les paramètres du spirographe dans Spirograph.fs correspondent à ceux affichés dans cette boîte de dialogue. Notez qu'il n'y a pas de vérification d'erreur: Si vous entrez un nombre à virgule flottante pour les entiers attendus dans les deux champs de paramètres supérieurs, le programme se bloque. Donc, s'il vous plaît ajouter une vérification d'erreur dans votre propre effort.

Notez également que les champs de saisie des paramètres sont désactivés avec la couleur aléatoire sélectionnée dans les boutons radio. C'est juste pour montrer comment cela peut être fait.

Afin de déplacer les données entre la boîte de dialogue et le programme, j'utilise le System.Windows.Element.FindName () pour trouver le contrôle approprié, le placer dans le contrôle approprié, puis obtenir les paramètres appropriés du fichier. Contrôle. La plupart des autres exemples de programmes utilisent la liaison de données. Je ne l'ai pas fait pour deux raisons: premièrement, je n'arrivais pas à comprendre comment le faire fonctionner, et deuxièmement, quand cela ne fonctionnait pas, je n'ai reçu aucun message d'erreur. Peut-être que quelqu'un qui visite ceci sur StackOverflow peut me dire comment utiliser la liaison de données sans inclure un nouvel ensemble de packages NuGet.

Ajoutez le code pour 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
```

Il n'y a pas grand chose à faire ici: nous ouvrons la boîte de dialogue Paramètres si nécessaire et nous avons la possibilité de redessiner le spirographe avec les paramètres actuels.

Ajoutez les fichiers App.xaml et App.xaml.fs pour tout relier

```
<!-- 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>
```

Voici le code derrière:

```
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 est tout à fait normal ici, principalement pour montrer où les ressources d'application, telles que les icônes, les graphiques ou les fichiers externes, peuvent être déclarées. Le compagnon App.xaml.fs regroupe le modèle et la fenêtre principale, dimensionne la fenêtre principale à deux tiers de la taille d'écran disponible et l'exécute.

Lorsque vous créez cela, n'oubliez pas de vous assurer que la propriété Build pour chaque fichier xaml est définie sur Resource. Vous pouvez ensuite exécuter le débogueur ou compiler un fichier exe. Notez que vous ne pouvez pas l'exécuter avec l'interpréteur F #: le package FsXaml et l'interpréteur sont incompatibles.

Voilà. J'espère que vous pourrez l'utiliser comme point de départ pour vos propres applications et, ce faisant, étendre vos connaissances au-delà de ce qui est montré ici. Tous les commentaires et suggestions seront appréciés.

Lire En utilisant F #, WPF, FsXaml, un menu et une boîte de dialogue en ligne: https://riptutorial.com/fr/fsharp/topic/9145/en-utilisant-f-sharp--wpf--fsxaml--un-menu-et-une-boite-de-dialogue

Chapitre 9: Évaluation paresseuse

Examples

Évaluation paresseuse Introduction

La plupart des langages de programmation, y compris F #, évaluent les calculs immédiatement en accord avec un modèle appelé Strict Evaluation. Cependant, dans l'évaluation différée, les calculs ne sont pas évalués tant qu'ils ne sont pas nécessaires. F # nous permet d'utiliser l'évaluation paresseuse à la fois par le mot clé et les `sequences` `lazy` .

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

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

De plus, lors de l'utilisation de l'évaluation différée, les résultats du calcul sont mis en cache, donc si nous forçons le résultat après notre première instance de forçage, l'expression elle-même ne sera plus évaluée.

```
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()
```

Introduction à l'évaluation paresseuse en F

F #, comme la plupart des langages de programmation, utilise Strict Evaluation par défaut. En évaluation stricte, les calculs sont exécutés immédiatement. En revanche, l'évaluation différée diffère l'exécution des calculs jusqu'à ce que leurs résultats soient nécessaires. De plus, les résultats d'un calcul sous évaluation différée sont mis en cache, évitant ainsi la nécessité de réévaluer une expression.

Nous pouvons utiliser l'évaluation paresseuse dans F # à la fois par le mot-clé `lazy` et les `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()
```

Lire Évaluation paresseuse en ligne: https://riptutorial.com/fr/fsharp/topic/3682/evaluation-paresseuse

Chapitre 10: Extensions de type et de module

Remarques

Dans tous les cas d'extension de types et de modules, le code d'extension doit être ajouté / chargé avant le code à appeler. Il doit également être mis à la disposition du code appelant en ouvrant / important les espaces de noms correspondants.

Examples

Ajout de nouvelles méthodes / propriétés aux types existants

F # permet d'ajouter des fonctions en tant que "membres" aux types lorsqu'ils sont définis (par exemple, Types d'enregistrement). Cependant, F # permet également d'ajouter de nouveaux membres d'instance *aux* types *existants* , même ceux déclarés ailleurs et dans d'autres langues .net.

L'exemple suivant ajoute une nouvelle méthode d'instance `Duplicate` à toutes les instances de `String` .

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

Remarque : `this` s'agit d'un nom de variable choisi arbitrairement pour désigner l'instance du type en cours d'extension - `x` fonctionnerait tout aussi bien, mais serait peut-être moins auto-descriptif.

Il peut alors être appelé de la manière suivante.

```
// 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
```

Cette fonctionnalité est très similaire aux méthodes d'extension en C #.

De nouvelles propriétés peuvent également être ajoutées aux types existants de la même manière. Ils deviendront automatiquement des propriétés si le nouveau membre ne prend aucun argument.

```
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
```

Ajout de nouvelles fonctions statiques aux types existants

F # permet d'étendre les types existants avec de nouvelles fonctions statiques.

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

Cette nouvelle fonction peut être appelée comme ceci:

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

Cette fonctionnalité peut signifier qu'au lieu de créer des bibliothèques de fonctions "utilitaires", elles peuvent être ajoutées aux types existants pertinents. Cela peut être utile pour créer plus de versions faciles à utiliser des fonctions permettant le 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
```

Ajout de nouvelles fonctions aux modules et types existants à l'aide de modules

Les modules peuvent être utilisés pour ajouter de nouvelles fonctions aux modules et aux types existants.

```
namespace FSharp.Collections

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

La nouvelle fonction peut alors être appelée comme si elle était un membre original de List.

```
open FSharp.Collections

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

Lire Extensions de type et de module en ligne:

https://riptutorial.com/fr/fsharp/topic/2977/extensions-de-type-et-de-module

Chapitre 11: F # Performance Tips and Tricks

Examples

Utiliser tail-recursion pour une itération efficace

Venant de langages impératifs, de nombreux développeurs se demandent comment écrire une `for-loop` qui se termine plus tôt, car `F#` ne supporte pas le `break` , le `continue` ou le `return` . La réponse dans `F#` est d'utiliser la récursivité, qui est une manière flexible et idiomatique d'itérer tout en offrant d'excellentes performances.

Disons que nous voulons implémenter `tryFind` pour `List` . Si `F#` charge le `return` nous écrirons `tryFind` un peu comme ceci:

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

Cela ne fonctionne pas dans `F#` . Au lieu de cela, nous écrivons la fonction en utilisant récursivité:

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

La récursion est performante dans `F#` car, lorsque le compilateur `F#` détecte qu'une fonction est récursive, il réécrit la récursivité en une `while-loop` efficace. En utilisant `ILSpy` nous pouvons voir que cela est vrai pour notre `loop` fonctions:

```
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;
}
```

En dehors de certaines affectations inutiles (qui, espérons-le, le JIT-er élimine), il s'agit essentiellement d'une boucle efficace.

De plus, la récursion de la queue est idiomatique pour `F#` car elle nous permet d'éviter un état mutable. Considérons une fonction de `sum` qui `sum` tous les éléments d'une `List` . Un premier essai évident serait ceci:

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

Si nous réécrivons la boucle en queue récursive, nous pouvons éviter l'état mutable:

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

Pour plus d'efficacité, le compilateur `F#` transforme cela en une `while-loop` qui utilise l'état mutable.

Mesurer et vérifier vos hypothèses de performance

Cet exemple est écrit avec `F#` en tête mais les idées sont applicables dans tous les environnements

La première règle lors de l'optimisation des performances est de ne pas s'appuyer sur des hypothèses. Toujours mesurer et vérifier vos hypothèses.

Comme nous n'écrivons pas directement le code machine, il est difficile de prédire comment le compilateur et JIT: er transforment votre programme en code machine. C'est pourquoi nous devons mesurer le temps d'exécution pour constater que nous obtenons l'amélioration des performances attendue et vérifier que le programme réel ne contient pas de surcharge cachée.

La vérification est le processus très simple qui implique le reverse engineering de l'exécutable en utilisant par exemple des outils tels que ILSpy . Le JIT: er complique la vérification en ce sens que la visualisation du code machine réel est délicate mais faisable. Cependant, l'examen du `IL-code` donne généralement de gros avantages.

Le problème le plus difficile est la mesure; plus difficile car il est difficile de mettre en place des situations réalistes permettant de mesurer les améliorations du code. Still Measuring est inestimable.

Analyser des fonctions simples F #

Examinons quelques fonctions simples `F#` qui accumulent tous les entiers dans `1..n` écrits de différentes manières. Comme la gamme est une série arithmétique simple, le résultat peut être calculé directement, mais pour les besoins de cet exemple, nous parcourons la plage.

Tout d'abord, nous définissons des fonctions utiles pour mesurer le temps nécessaire à une fonction:

```
// 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` exécute une action à plusieurs reprises, nous devons exécuter les tests pendant quelques centaines de millisecondes pour réduire les écarts.

Ensuite, nous définissons quelques fonctions qui accumulent tous les entiers dans `1..n` de différentes manières.

```
// 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
```

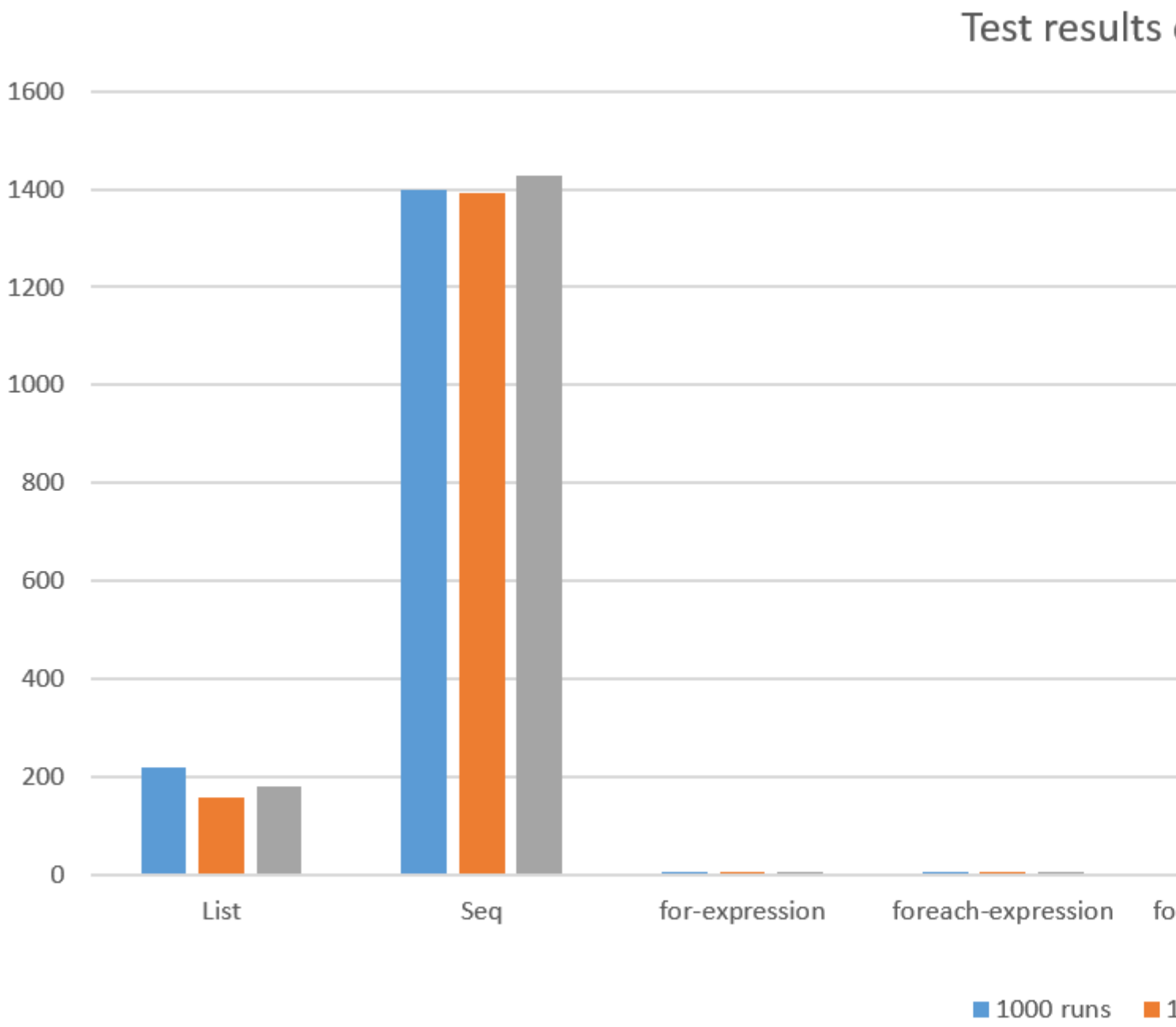
```
  for i in [1..n] do
    sum <- sum + i
  sum

// 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
```

Nous supposons que le résultat est le même (sauf pour l'une des fonctions qui utilise l'incrément de `2`), mais y a-t-il une différence de performance. Pour mesurer ceci, la fonction suivante est définie:

```
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
```

Le résultat du test lors de l'exécution sur .NET 4.5.2 x64:



Nous constatons une différence spectaculaire et certains résultats sont inattendus.

Regardons les mauvais cas:

liste

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

Ce qui se passe ici est une liste complète contenant tous les entiers `1..n` est créé et réduit en utilisant une somme. Cela devrait être plus coûteux que de simplement itérer et accumuler sur la plage, il semble environ ~ 42 fois plus lent que la boucle for.

De plus, nous pouvons voir que le GC a fonctionné environ 100 fois pendant le test, car le code

allouait beaucoup d'objets. Cela coûte aussi le processeur.

Seq

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

La version de `Seq` n'alloue pas une `List` complète donc c'est un peu surprenant que cette ~ 270x soit plus lente que la boucle for. De plus, on voit que le GC a exécuté 661x.

`Seq` est inefficace lorsque la quantité de travail par élément est très faible (dans ce cas, en agrégeant deux entiers).

Le but n'est pas de ne jamais utiliser `Seq` . Le point est de mesurer.

(**manofstick edit:** `Seq.init` est le coupable de ce grave problème de performances. Il est beaucoup plus efficace d'utiliser l'expression `{ 0 .. n }` au lieu de l' `Seq.init (n+1) id` . Cela deviendra beaucoup plus efficace encore. quand ce PR est fusionné et libéré, même après la publication, `Seq.init ... |> Seq.sum` sera toujours lent, mais quelque peu contre-intuitif, `Seq.init ... |> Seq.map id |> Seq.sum` sera assez rapide pour maintenir une compatibilité ascendante avec l' `Seq.init` de `Seq.init` , qui ne calcule pas initialement `Current` , mais les encapsule plutôt dans un objet `Lazy` - même si cela devrait être un peu meilleur en raison de ce PR Note à l'éditeur: désolé c'est une sorte de notes décousues, mais je ne veux pas que les gens soient rebutés par Seq quand l'amélioration est imminente ... *Quand cela arrivera, il serait bon de mettre à jour les graphiques qui sont sur cette page.*)

foreach-expression over List

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

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

La différence entre ces deux fonctions est très subtile mais la différence de performance n'est pas, à peu près ~ 76x. Pourquoi? Ingénérons le mauvais code:

```
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` est implémenté comme une boucle `while` efficace, mais `for i in [1..n]` est converti en:

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

Cela signifie d'abord que nous créons une `Seq` over `1..n` et que nous `ToList` finalement `ToList` .

Coûteux.

incrément d'expression de foreach 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
```

Encore une fois la différence entre ces deux fonctions est subtile mais la différence de performance est brutale: ~ 25x

Encore une fois, `ILSpy` :

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

Un `Seq` est créé sur `1..2..n` et ensuite nous `1..2..n` `Seq` utilisant l'énumérateur.

Nous nous attendions à ce que `F#` crée quelque chose comme ceci:

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

Cependant, `F#` compilateur `F#` ne prend en charge que les boucles efficaces sur les plages int32 incrémentées de un. Pour tous les autres cas, il se rabat sur `Operators.OperatorIntrinsics.RangeInt32` . Ce qui expliquera le prochain résultat surprenant

foreach-expression sur 64 bit

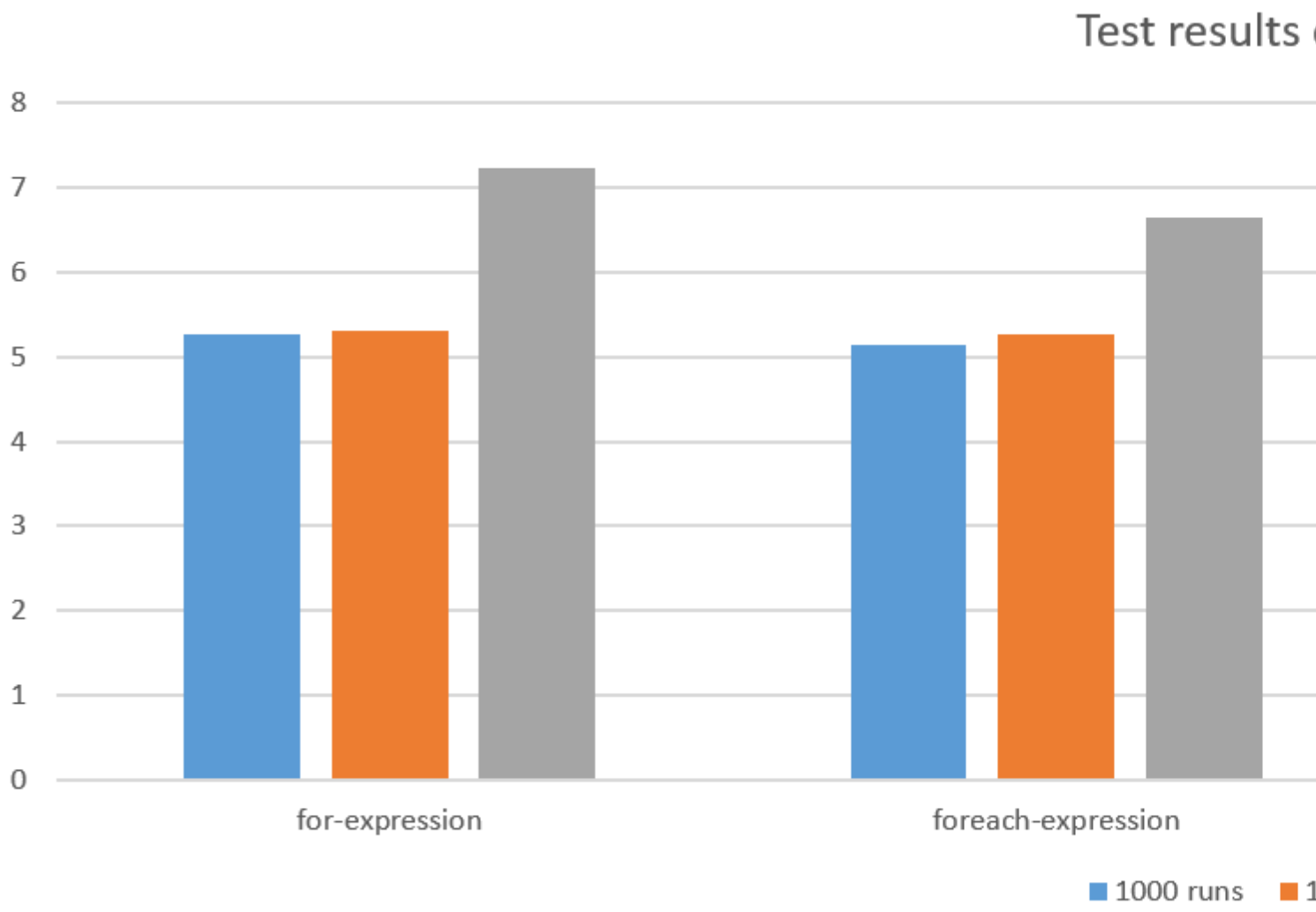
```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int
```

Cela effectue ~ 47x plus lentement que la boucle for, la seule différence est que nous parcourons des entiers de 64 bits. `ILSpy` nous montre pourquoi:

```
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#` ne prend en charge que les boucles efficaces pour les numéros `int32` qu'il doit utiliser les `Operators.OperatorIntrinsics.RangeInt64` secours.OperatorIntrinsics.RangeInt64.

Les autres cas sont à peu près similaires:



La raison pour laquelle les performances se dégradent pour les tests de plus grande envergure réside dans le fait que l'invocation de l' `action` fait de plus en plus au fur et à `action` que nous travaillons de moins en moins en `action` .

Faire une boucle vers `0` peut parfois apporter des avantages en termes de performances, car cela peut sauver un registre de CPU, mais dans ce cas, le CPU a des registres à épargner, cela ne semble donc pas faire de différence.

Conclusion

La mesure est importante car sinon nous pourrions penser que toutes ces alternatives sont équivalentes mais que certaines alternatives sont ~ 270x plus lentes que d'autres.

L'étape de vérification implique l'ingénierie inverse. L'exécutable nous aide à expliquer *pourquoi* nous avons obtenu ou non les performances attendues. De plus, la vérification peut nous aider à prédire la performance dans les cas où il est trop difficile d'effectuer une mesure correcte.

Il est difficile de prévoir les performances là-bas, toujours Mesurer, toujours vérifier vos hypothèses de performance.

Comparaison de différents pipelines de données F

Dans `F#` il existe de nombreuses options pour créer des pipelines de données, par exemple: `List` , `Seq` et `Array` .

Quel pipeline de données est préférable du point de vue de l'utilisation de la mémoire et des performances?

Afin de répondre à cette question, nous comparerons les performances et l'utilisation de la mémoire en utilisant différents pipelines.

Pipeline de données

Afin de mesurer les frais généraux, nous utiliserons un pipeline de données à faible coût par unité de traitement, par produit traité:

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

Nous allons créer des pipelines équivalents pour toutes les alternatives et les comparer.

Nous allons varier la taille de `n` mais laisser le nombre total de travail être le même.

Solutions de pipeline de données

Nous comparerons les alternatives suivantes:

1. Code impératif
2. Tableau (non paresseux)
3. Liste (non paresseux)
4. LINQ (flux de tirage paresseux)
5. Seq (flux de tirage paresseux)
6. Nessos (flux de traction / poussée paresseux)
7. PullStream (flux de tirage simpliste)
8. PushStream (flux de diffusion simpliste)

Bien que ce ne soit pas un pipeline de données, nous le comparerons au code `Imperative` car cela correspond le mieux à la manière dont le processeur exécute le code. Cela devrait être le moyen le plus rapide de calculer le résultat, ce qui nous permet de mesurer les performances des pipelines de données.

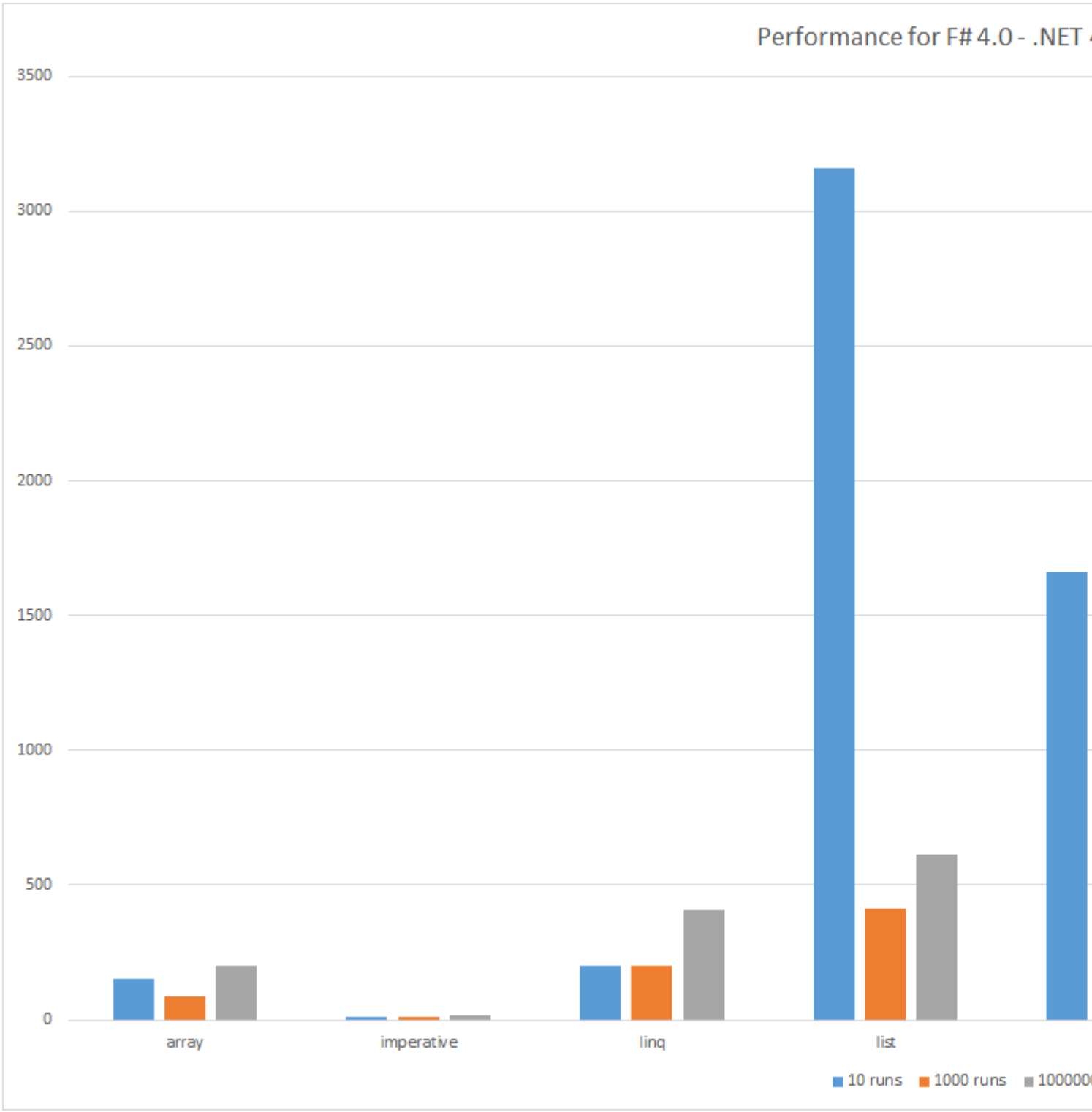
`Array` et `List` calculent un `Array` / `List` complet à chaque étape, de sorte que nous attendons une surcharge de mémoire.

`LINQ` et `Seq` sont tous deux basés sur `IEnumerable<'T>` qui est un flux d'extraction paresseux (pull signifie que le flux consommateur extrait des données du flux producteur). Nous nous attendons donc à ce que les performances et l'utilisation de la mémoire soient identiques.

`Nessos` est une bibliothèque de flux haute performance qui prend en charge à la fois la fonction push & pull (comme Java `Stream`).

PullStream et PushStream sont des implémentations simplistes des flux `Pull` & `Push` .

Résultats de performance de l'exécution sur: F # 4.0 - .NET 4.6.1 - x64

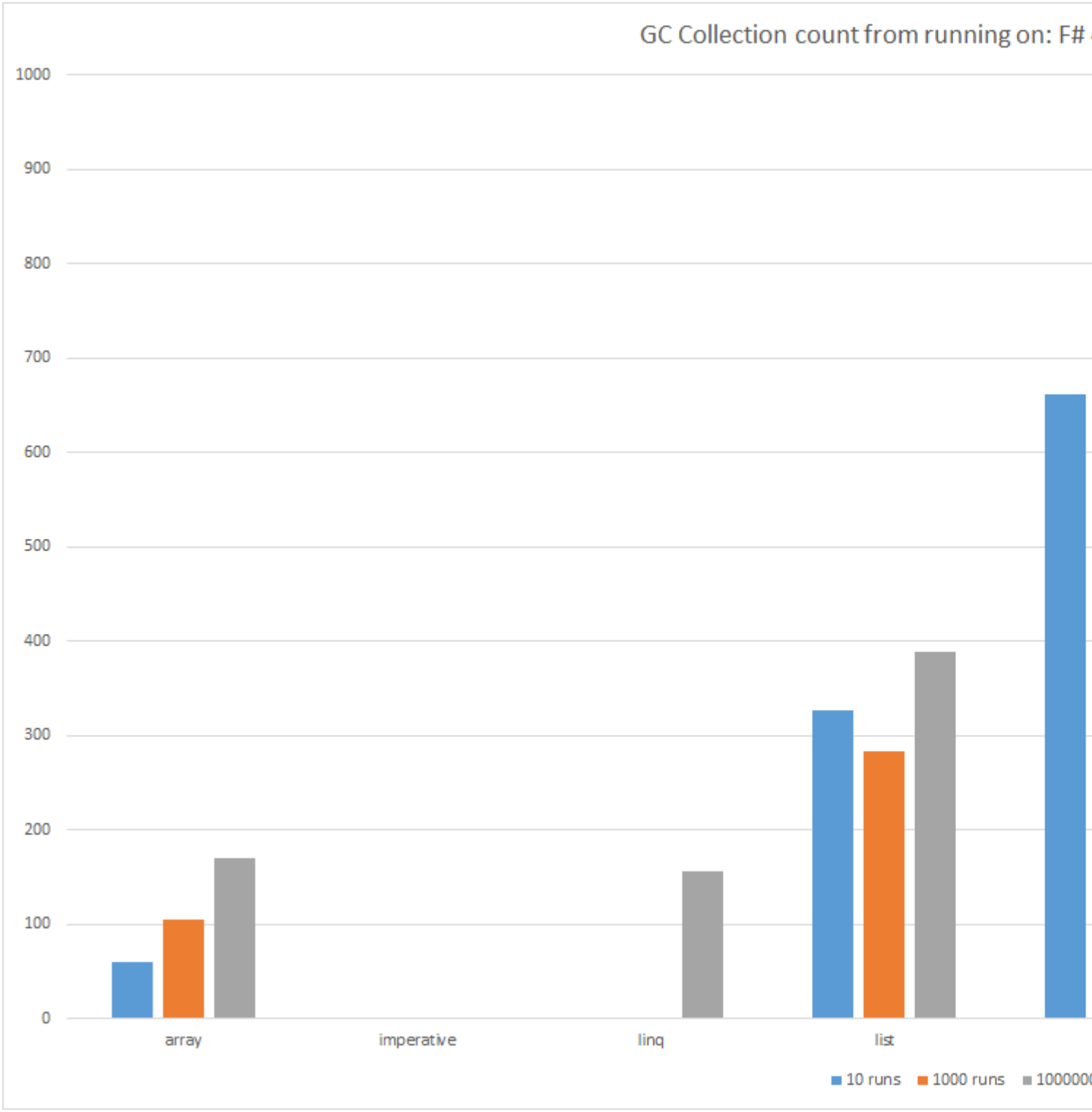


Les barres indiquent le temps écoulé, inférieur est meilleur. La quantité totale de travail utile est la même pour tous les tests, les résultats sont donc comparables. Cela signifie également que peu d'exécutions impliquent des ensembles de données plus importants.

Comme d'habitude lors de la mesure on voit des résultats intéressants.

1. `List` performances médiocres est comparée à d'autres alternatives pour les grands ensembles de données. Cela peut être dû à une `GC` ou à une mauvaise localisation de cache.
2. Performance du `Array` meilleure que prévu.
3. `LINQ` fonctionne mieux que `Seq` , ceci est inattendu car les deux sont basés sur `IEnumerable<'T>` . Cependant, `Seq` interne est basé sur une implémentation générique pour tous les algorithmes, tandis que `LINQ` utilise des algorithmes spécialisés.
4. `Push` effectue mieux que `Pull` . Cela est attendu car le pipeline de données push a moins de vérifications
5. Les pipelines de données `Push` simplistes sont comparables à `Nessos` . Cependant, `Nessos` soutient la traction et le parallélisme.
6. Pour les pipelines de données de petite taille, les performances de `Nessos` se dégradent car la configuration des pipelines entraîne une surcharge.
7. Comme prévu, le code `Imperative` le meilleur

GC Collection compte à partir de l'exécution sur: F # 4.0 - .NET 4.6.1 - x64



Les barres indiquent le nombre total de comptes de collection de `GC` pendant le test, inférieur est meilleur. Il s'agit d'une mesure du nombre d'objets créés par le pipeline de données.

Comme d'habitude lors de la mesure on voit des résultats intéressants.

1. `List` crée vraisemblablement plus d'objets que `Array` car une `List` est essentiellement une liste unique de nœuds liés. Un tableau est une zone de mémoire continue.
2. En regardant les chiffres sous-jacents, `List` et `Array` imposent des collections de 2 générations. Ce type de collection est cher.
3. `Seq` déclenche une quantité surprenante de collections. C'est étonnamment encore pire que

`List` à cet égard.

4. `LINQ` , `Nessos` , `Push` and `Pull` ne déclenchent aucune collecte pour quelques exécutions. Cependant, les objets sont alloués pour que le `GC` finisse par fonctionner.
5. Comme prévu, le code `Imperative` n'allouant aucun objet, aucune collection `GC` n'a été déclenchée.

Conclusion

Tous les pipelines de données effectuent la même quantité de travail utile dans tous les cas de test, mais nous constatons des différences significatives dans les performances et l'utilisation de la mémoire entre les différents pipelines.

En outre, nous remarquons que la surcharge des pipelines de données diffère en fonction de la taille des données traitées. Par exemple, pour les petites tailles, `Array` fonctionne assez bien.

Il ne faut pas oublier que la quantité de travail effectuée à chaque étape du pipeline est très faible pour pouvoir mesurer les frais généraux. Dans des situations "réelles", la surcharge de `Seq` peut ne pas avoir d'importance, car le travail prend plus de temps.

Les différences d'utilisation de la mémoire sont plus préoccupantes. `GC` n'est pas libre et il est bénéfique pour les applications de longue durée de maintenir la pression du `GC` basse.

Pour les développeurs `F#` concernés par les performances et l'utilisation de la mémoire, il est recommandé de vérifier Nessos Streams .

Si vous avez besoin de performances de haut niveau placées stratégiquement, le code `Imperative` mérite d'être considéré.

Enfin, en matière de performance, ne faites pas d'hypothèses. Mesurer et vérifier.

Code source complet:

```
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
```

Lire F # Performance Tips and Tricks en ligne: https://riptutorial.com/fr/fsharp/topic/3562/f-sharp-performance-tips-and-tricks

Chapitre 12: F # sur .NET Core

Examples

Créer un nouveau projet via l'interface CLI de dotnet

Une fois les outils .NET CLI installés, vous pouvez créer un nouveau projet avec la commande suivante:

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

Cela crée un programme en ligne de commande.

Flux de travail initial du projet

Créer un nouveau projet

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

Restaurez tous les packages répertoriés dans project.json

```
dotnet restore
```

Un fichier project.lock.json doit être écrit.

Exécuter le programme

```
dotnet run
```

Ce qui précède compilera le code si nécessaire.

La sortie du projet par défaut créé par `dotnet new -lf#` contient les éléments suivants:

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

Lire F # sur .NET Core en ligne: https://riptutorial.com/fr/fsharp/topic/4404/f-sharp-sur--net-core

Chapitre 13: Fournisseurs de type

Examples

Utilisation du fournisseur de type CSV

Compte tenu du fichier CSV suivant:

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

Vous pouvez lire les données avec le script suivant:

```
#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
```

Utilisation du fournisseur de type WMI

Le fournisseur de type WMI vous permet d'interroger les services WMI avec un typage fort.

Pour générer les résultats d'une requête WMI en tant que 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)
```

Lire Fournisseurs de type en ligne: https://riptutorial.com/fr/fsharp/topic/1631/fournisseurs-de-type

Chapitre 14: Génériques

Examples

Inversion d'une liste de tout type

Pour inverser une liste, il n'est pas important de savoir quel type sont les éléments de la liste, mais uniquement leur ordre. C'est un candidat idéal pour une fonction générique, donc la même fonction d'inversion peut être utilisée quelle que soit la liste transmise.

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

Le code ne fait aucune hypothèse sur les types d'éléments. Le compilateur (ou F # interactif) vous dira que la signature de type de cette fonction est `'T list -> 'T list` Le `'T` vous dit que c'est un type générique sans contraintes. Vous pouvez également voir `'a` au lieu de `'T` - la lettre est sans importance car ce n'est qu'un espace réservé *générique* . Nous pouvons passer une `int list` ou une `string list` , et les deux fonctionneront correctement, renvoyant respectivement une `int list` ou une `string list` .

Par exemple, dans F # interactive:

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

Mapper une liste dans un type différent

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

La signature de cette fonction est `('a -> 'b) -> 'a list -> 'b list` , ce qui est le plus générique possible. Cela n'empêche pas `'a` d'être du même type que d'être `'b` , mais cela leur permet également d'être différent. Ici, vous pouvez voir que le `'a` type qui est le paramètre de la fonction `f` doit correspondre au type du paramètre `list` . Cette fonction est toujours générique, mais il existe de légères contraintes sur les entrées - si les types ne correspondent pas, il y aura une erreur de compilation.

Exemples:

```
> 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'
```

Lire Génériques en ligne: https://riptutorial.com/fr/fsharp/topic/7731/generiques

Chapitre 15: Implémentation d'un modèle de conception dans F

Examples

Programmation pilotée par les données en F

Grâce à l'inférence de type et à l'application partielle dans `F#` la programmation pilotée par les données est succincte et lisible.

Imaginons que nous vendions une assurance automobile. Avant d'essayer de le vendre à un client, nous essayons de déterminer si le client est un client potentiel valide pour notre entreprise en vérifiant le sexe et l'âge du client.

Un modèle client simple:

```
type Sex =
  | Male
  | Female

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

Ensuite, nous voulons définir une liste d'exclusion (tableau) afin que si un client correspond à une ligne de la liste d'exclusion, le client ne peut pas acheter notre assurance automobile.

```
//  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
  |]
```

En raison de l'inférence de type et de l'application partielle, la liste d'exclusion est flexible mais facile à comprendre.

Enfin, nous définissons une fonction qui utilise la liste d'exclusion (une table) pour séparer les clients en deux compartiments: clients potentiels et refusés.

```
// 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 ()
```

Pour conclure, définissons certains clients et voyons s'ils sont des clients potentiels pour notre assurance automobile parmi eux:

```
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
```

Cela 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;})|]
```

Lire Implémentation d'un modèle de conception dans F # en ligne: https://riptutorial.com/fr/fsharp/topic/3925/implementation-d-un-modele-de-conception-dans-f-sharp

Chapitre 16: Introduction à WPF en F

Introduction

Cette rubrique illustre comment exploiter la **programmation fonctionnelle** dans une **application WPF** . Le premier exemple provient d'un article de Māris Krivtežs (voir la section *Remarques* en bas). La raison de revoir ce projet est double:

1 \ La conception prend en charge la séparation des problèmes, tandis que le modèle reste pur et que les modifications sont propagées de manière fonctionnelle.

2 \ La ressemblance permettra une transition facile vers la mise en œuvre de Gjallarhorn.

Remarques

Projets de démonstration de la bibliothèque @GitHub

- FSharp.ViewModule (sous FsXaml)
- Gjallarhorn (réf Samples)

M. Krivtežs a écrit deux excellents articles sur ce sujet:

- Application F # XAML - MVVM vs MVC où les avantages et les inconvénients des deux approches sont mis en évidence.

  Je pense qu'aucun de ces styles d'application XAML ne profite beaucoup de la programmation fonctionnelle. J'imagine que l'application idéale serait la vue qui produit des événements et les événements contiennent l'état d'affichage actuel. Toute la logique de l'application doit être traitée en filtrant et en manipulant les événements et le modèle de vue. Dans la sortie, elle doit produire un nouveau modèle de vue lié à la vue.

- F # XAML - MVVM événementiel revisité dans le sujet ci-dessus.

Examples

FSharp.ViewModule

Notre application de démonstration consiste en un tableau de bord. Le modèle de score est un enregistrement immuable. Les événements du tableau d'affichage sont contenus dans un type d'union.

```
namespace Score.Model

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

Les modifications sont propagées en écoutant les événements et en mettant à jour le modèle de vue en conséquence. Au lieu d'ajouter des membres au type de modèle, comme dans la POO, nous déclarons un module distinct pour héberger les opérations autorisées.

```
[<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
```

Notre modèle de vue dérive de `EventViewModelBase<'a>` , qui possède une propriété `EventStream` de type `IObservable<'a>` . Dans ce cas, les événements auxquels nous souhaitons souscrire sont de type `ScoringEvent` .

Le contrôleur gère les événements de manière fonctionnelle. Sa signature `Score -> ScoringEvent -> Score` nous montre que chaque fois qu'un événement se produit, la valeur actuelle du modèle est transformée en une nouvelle valeur. Cela permet à notre modèle de rester pur, même si notre modèle de vue ne l'est pas.

Un `eventHandler` est chargé de modifier l'état de la vue. Héritage de `EventViewModelBase<'a>` nous pouvons utiliser `EventValueCommand` et `EventValueCommandChecked` pour connecter les événements aux commandes.

```
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
```

Le code derrière le fichier (* .xaml.fs) est l'endroit où tout est mis en place, c'est-à-dire que la

fonction de mise à jour (`controller`) est injectée dans `MainViewModel` .

```
namespace Score.Views

open FsXaml

type MainView = XAML<"MainWindow.xaml">

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

Le type `CompositionRoot` sert de wrapper référencé dans le fichier XAML.

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

Je ne vais pas plonger plus profondément dans le fichier XAML car ce sont des choses WPF de base, le projet entier se trouve sur GitHub .

Gjallarhorn

Les principaux types de la bibliothèque Gjallarhorn implémentent `IObservable<'a>` , ce qui rendra l'implémentation plus familière (souvenez-vous de la propriété `EventStream` de l'exemple FSharp.ViewModule). Le seul changement réel à notre modèle est l'ordre des arguments de la fonction de mise à jour. De plus, nous utilisons maintenant le terme *Message* au lieu de *Event* .

```
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
```

Afin de créer une interface utilisateur avec Gjallarhorn, au lieu de créer des classes pour prendre en charge la liaison de données, nous créons des fonctions simples appelées `Component` . Dans leur constructeur, le premier argument `source` est de type `BindingSource` (défini dans Gjallarhorn.Bindable) et permet de mapper le modèle à la vue et les événements de la vue dans les messages.

```
namespace ScoreLogic.Model

open Gjallarhorn
open Gjallarhorn.Bindable

module Program =

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

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

L'implémentation actuelle diffère de la version de FSharp.ViewModule dans la mesure où CanExecute n'est pas encore correctement implémenté dans deux commandes. Liste également la plomberie de l'application.

```
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
```

À gauche avec la configuration de la vue découplée, en combinant le type `MainWindow` et l'application logique.

```
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
```

Cela résume les concepts de base, pour plus d'informations et un exemple plus élaboré, veuillez vous reporter au message de Reed Copsey . Le projet des *arbres de Noël* met en évidence quelques avantages à cette approche:

- En nous remplaçant efficacement de la nécessité de copier (manuellement) le modèle dans une collection personnalisée de modèles de vue, en les gérant et en reconstruisant manuellement le modèle à partir des résultats.
- Les mises à jour au sein des collections sont effectuées de manière transparente, tout en conservant un modèle pur.
- La logique et la vue sont hébergées par deux projets différents, mettant l'accent sur la séparation des préoccupations.

Lire Introduction à WPF en F # en ligne: https://riptutorial.com/fr/fsharp/topic/8758/introduction-a-wpf-en-f-sharp

Chapitre 17: Le type "unit"

Examples

À quoi sert un tuple 0?

Un 2-tuple ou un 3-tuple représentent un groupe d'éléments apparentés. (Points dans l'espace 2D, valeurs RVB d'une couleur, etc.) Un 1-tuple n'est pas très utile car il pourrait facilement être remplacé par un seul `int` .

Un 0-tuple semble encore plus inutile puisqu'il ne contient absolument *rien* . Pourtant, il a des propriétés qui le rendent très utile dans les langages fonctionnels comme F #. Par exemple, le type 0-tuple a exactement *une* valeur, généralement représentée par `()` . Tous les 0-tuples ont cette valeur, c'est donc essentiellement un type singleton. Dans la plupart des langages de programmation fonctionnels, y compris F #, cela s'appelle le type d' `unit` .

Les fonctions qui renvoient un `void` dans C # retourneront le type d' `unit` dans F #:

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

Exécutez cela dans l'interpréteur interactif F # et vous verrez:

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

Cela signifie que la valeur `printResult` est de type `unit` et a la valeur `()` (le tuple vide, la seule et unique valeur du type d' `unit`).

Les fonctions peuvent également prendre le type d' `unit` comme paramètre. En F #, les fonctions peuvent sembler ne prendre aucun paramètre. Mais en fait, ils prennent un seul paramètre de type `unit` . Cette fonction:

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

est en fait équivalent à:

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

C'est-à-dire une fonction qui prend un paramètre de type `unit` et renvoie la valeur `int` 6. Si vous regardez la signature de type que l'interpréteur interactif F # imprime lorsque vous définissez cette fonction, vous verrez:

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

Le fait que toutes les fonctions prennent au moins un paramètre et renvoie une valeur, même si cette valeur est parfois une valeur "inutile" comme `()` , signifie que la composition de la fonction

est beaucoup plus facile dans F # que dans les langues qui type d' `unit` . Mais c'est un sujet plus avancé auquel nous reviendrons plus tard. Pour l'instant, rappelez-vous simplement que lorsque vous voyez l' `unit` dans une signature de fonction, ou `()` dans les paramètres d'une fonction, c'est le type 0-tuple qui sert à dire "Cette fonction prend ou retourne aucune valeur significative."

Différer l'exécution du code

Nous pouvons utiliser le type d' `unit` comme argument de fonction pour définir des fonctions que nous ne voulons pas exécuter plus tard. Ceci est souvent utile dans les tâches d'arrière-plan asynchrones, lorsque le thread principal peut vouloir déclencher certaines fonctionnalités prédéfinies du thread d'arrière-plan, comme le déplacer dans un nouveau fichier ou si une let-binding ne doit pas être exécutée immédiatement:

```
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)
```

Dans le code suivant, nous définissons le code pour démarrer un "worker" qui imprime simplement la valeur sur laquelle il travaille toutes les 2 secondes. Le travailleur retourne alors deux fonctions qui peuvent être utilisées pour le contrôler - une qui le déplace vers la valeur suivante sur laquelle travailler et une qui l'empêche de fonctionner. Celles-ci doivent être des fonctions, car nous ne voulons pas que leurs corps soient exécutés jusqu'à ce que nous choisissions de le faire, sinon le travailleur passerait immédiatement à la seconde valeur et s'arrêterait sans avoir rien fait.

```
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
```

Nous pouvons alors démarrer un travailleur en faisant

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

et le travail va commencer - si nous sommes en F # interactif, nous verrons les messages imprimés dans la console toutes les deux secondes. Nous pouvons alors courir

```
nextValue ()
```

et nous verrons les messages indiquant que la valeur en cours de traitement est passée à la suivante.

Quand il est temps de finir de travailler, nous pouvons exécuter le

```
stopOnNextTick ()
```

fonction, qui va imprimer le message de fermeture, puis quitter.

Le type d' `unit` est important ici pour signifier "aucune entrée" - les fonctions ont déjà toutes les informations dont elles ont besoin pour travailler avec elles, et l'appelant n'est pas autorisé à changer cela.

Lire Le type "unit" en ligne: https://riptutorial.com/fr/fsharp/topic/2513/le-type--unit-

Chapitre 18: Les fonctions

Examples

Fonctions de plusieurs paramètres

En F #, **toutes les fonctions prennent exactement un paramètre** . Cela semble une déclaration étrange, car il est trivialement facile de déclarer plus d'un paramètre dans une déclaration de fonction:

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

Mais si vous tapez cette déclaration de fonction dans l'interpréteur interactif F #, vous verrez que sa signature de type est la suivante:

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

Sans les noms de paramètres, cette signature est `int -> int -> int` . L'opérateur `->` est associatif à droite, ce qui signifie que cette signature est équivalente à `int -> (int -> int)` . En d'autres termes, `add` est une fonction qui prend un paramètre `int` et renvoie **une fonction qui prend un `int` et renvoie `int`** . Essayez-le:

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

Cependant, vous pouvez également appeler une fonction comme `add` d'une manière plus "conventionnelle", en lui passant directement deux paramètres, et cela fonctionnera comme prévu:

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

Cela s'applique aux fonctions avec autant de paramètres que vous le souhaitez:

```
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
```

Cette méthode de réflexion sur les fonctions multi-paramètres est une fonction qui prend un paramètre et retourne de nouvelles fonctions (qui peuvent à leur tour prendre un paramètre et

renvoyer de nouvelles fonctions jusqu'à ce que la fonction finale prenne le paramètre final et retourne un résultat) est appelé **currying** , en l'honneur du mathématicien Haskell Curry, célèbre pour avoir développé le concept. (Il a été inventé par quelqu'un d'autre, mais Curry mérite à juste titre le mérite.)

Ce concept est utilisé dans tout F # et vous voudrez bien le connaître.

Bases des fonctions

La plupart des fonctions de F # sont créées avec la syntaxe `let` :

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

Ceci définit une fonction nommée `timesTwo` qui prend un seul paramètre `x` . Si vous exécutez une session F # interactive (`fsharpi` sous OS X et Linux, `fsi.exe` sous Windows) et collez cette fonction dans (et ajoutez le `;;` indiquant à `fsharpi` d'évaluer le code que vous venez de saisir), vous verrez que répond avec:

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

Cela signifie que `timesTwo` est une fonction qui prend un seul paramètre `x` de type `int` et retourne un `int` . Les signatures de fonctions sont souvent écrites sans les noms de paramètres, vous verrez donc souvent ce type de fonction écrit en tant que `int -> int` .

Mais attendez! Comment F # a-t-il su que `x` était un paramètre entier, puisque vous n'avez jamais spécifié son type? Cela est dû à l' **inférence de type** . Parce que dans le corps de la fonction, vous avez multiplié `x` par `2` , les types de `x` et `2` doivent être identiques. (En règle générale, F # ne convertira pas implicitement des valeurs en différents types; vous devez spécifier explicitement toutes les typographies souhaitées).

Si vous voulez créer une fonction qui ne prend pas de paramètre, c'est la **mauvaise** manière de le faire:

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

La **bonne** façon de le faire est la suivante:

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

Cette fonction `hello` a le type `unit -> unit` , qui est expliqué dans le type "unit" .

Fonctions curry vs tupled

Il existe deux manières de définir des fonctions avec plusieurs paramètres dans F #, fonctions Curry et fonctions 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
```

Toutes les fonctions définies en dehors de F # (comme le framework .NET) sont utilisées en F # avec la forme Tupled. La plupart des fonctions des modules de base F # sont utilisées avec la forme Curry.

La forme curry est considérée comme idiomatique F #, car elle permet une application partielle. Aucun des deux exemples suivants n'est possible avec la forme Tupled.

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

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

La raison en est que la fonction Curry, lorsqu'elle est appelée avec un paramètre, renvoie une fonction. Bienvenue dans la programmation fonctionnelle !!

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

Vous pouvez voir que c'est exactement la même signature.

Toutefois, lors de l’interfaçage avec d’autres codes .NET, comme lors de l’écriture de bibliothèques, il est important de définir des fonctions à l’aide de la forme Tupled.

Inlining

Inlining vous permet de remplacer un appel à une fonction par le corps de la fonction.

Cela est parfois utile pour des raisons de performances sur une partie critique du code. Mais la contrepartie est que votre assemblage prendra beaucoup de place puisque le corps de la fonction est dupliqué partout où un appel a eu lieu. Vous devez faire attention lorsque vous décidez d'inclure ou non une fonction.

Une fonction peut être insérée avec le mot-clé `inline` :

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

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

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

Un autre exemple avec une valeur locale:

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

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

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

Tuyau avant et arrière

Les opérateurs de tuyauterie sont utilisés pour transmettre des paramètres à une fonction de manière simple et élégante. Il permet d'éliminer les valeurs intermédiaires et facilite la lecture des appels de fonctions.

En F #, il y a deux opérateurs de tuyauterie:

- **Transférer** (`|>`): Passer des paramètres de gauche à droite

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

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

- **Arrière** (`<|`): Passage des paramètres de droite à gauche

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

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

Voici un exemple sans opérateur de pipe:

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

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

printfn "%A" finalSeq
```

Nous pouvons raccourcir l'exemple précédent et le rendre plus propre avec l'opérateur du tube

avant:

```
// 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
```

Chaque résultat de fonction est passé en paramètre à la fonction suivante.

Si vous souhaitez transmettre plusieurs paramètres à l'opérateur du canal, vous devez ajouter un `|` pour chaque paramètre supplémentaire et créer un tuple avec les paramètres. L'opérateur de canal natif F # prend en charge jusqu'à trois paramètres (|||> ou <|||).

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

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

Lire Les fonctions en ligne: https://riptutorial.com/fr/fsharp/topic/2525/les-fonctions

Chapitre 19: Les opérateurs

Examples

Comment composer des valeurs et des fonctions en utilisant des opérateurs communs

En programmation orientée objet, une tâche courante consiste à composer des objets (valeurs). Dans la programmation fonctionnelle, il est courant de composer des valeurs et des fonctions.

Nous sommes habitués à composer des valeurs à partir de notre expérience d'autres langages de programmation en utilisant des opérateurs tels que `+` , `-` , `*` , `/` etc.

Composition de la valeur

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

Comme la programmation fonctionnelle compose à la fois des fonctions et des valeurs, il n'est pas surprenant qu'il existe des opérateurs communs pour la composition de fonctions, tels que `>>` , `<<` , `|>` et `<|` .

Composition de fonction

```
// 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#` , la tuyauterie avant est préférable à la tuyauterie inversée car:

1. L'inférence de type (généralement) circule de gauche à droite, il est donc naturel que les valeurs et les fonctions circulent également de gauche à droite
2. Parce que `<|` et `<<` devraient avoir une associativité correcte mais en `F#` ils sont laissés associatifs ce qui nous oblige à insérer ()
3. Mélanger la tuyauterie avant et arrière ne fonctionne généralement pas car ils ont la même priorité.

Composition monade

Comme les monades (comme `Option<'T>` ou `List<'T>`) sont couramment utilisées dans la programmation fonctionnelle, il existe également des opérateurs courants mais moins connus pour composer des fonctions fonctionnant avec des monades comme `>>=` , `>=>` , `<|>` et `<*>` .

```
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
```

Conclusion

Pour les nouveaux programmeurs fonctionnels, la composition des fonctions à l'aide d'opérateurs peut sembler opaque et obscure, mais c'est parce que la signification de ces opérateurs n'est pas connue sous le nom d'opérateurs travaillant sur des valeurs. Cependant, avec un entraînement utilisant `|>` , `>>` , `>>=` et `>=>` devient aussi naturel que d'utiliser `+` , `-` , `*` et `/` .

Reliure tardive en F # en utilisant? opérateur

Dans un langage typé statiquement comme `F#` nous travaillons avec des types connus au moment de la compilation. Nous consommons des sources de données externes de manière sécurisée en utilisant des fournisseurs de type.

Cependant, il est parfois nécessaire d'utiliser une liaison tardive (comme la `dynamic` en `C#`). Par exemple, lorsque vous travaillez avec `JSON` documents `JSON` qui n'ont pas de schéma bien défini.

Pour simplifier le travail avec une liaison tardive, `F#` prend en charge les opérateurs de recherche

dynamique `?` et `?<-` .

Exemple:

```
// (?) 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
```

Il s'avère que le support `F#` pour la liaison tardive est simple mais flexible.

Lire Les opérateurs en ligne: https://riptutorial.com/fr/fsharp/topic/4641/les-operateurs

Chapitre 20: Les types

Examples

Introduction aux types

Les types peuvent représenter différents types de choses. Il peut s'agir d'une seule donnée, d'un ensemble de données ou d'une fonction.

En F #, nous pouvons regrouper les types en deux catégories .:

- Types 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
```

- Types .NET
    - Type intégré (int, bool, string, ...)
    - Classes, structures et interfaces
    - Les délégués
    - Tableaux

Abréviations de type

Les abréviations de type vous permettent de créer des alias sur des types existants pour leur donner un sens plus significatif.

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

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

Ensuite, vous pouvez utiliser l'alias comme n'importe quel autre type:

```
// 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";}
```

Attention, les alias ne vérifient pas la cohérence des types. Cela signifie que deux alias ciblant le même type peuvent être affectés l'un à l'autre:

```
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";}
```

Les types sont créés en F # en utilisant le mot-clé type

`F#` utilise le mot-clé `type` pour créer différents types de types.

1. Alias de type
2. Types de syndicats discriminés
3. Types d'enregistrement
4. Types d'interface
5. Types de classes
6. Types de structures

Exemples avec code `C#` équivalent lorsque possible:

```
// 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
```

Type Inférence

Reconnaissance

Cet exemple est adapté de cet article sur l' inférence de type

Qu'est-ce que le type Inference?

Type Inference est le mécanisme qui permet au compilateur de déduire quels types sont utilisés et où. Ce mécanisme repose sur un algorithme souvent appelé «Hindley-Milner» ou «HM». Voir ci-dessous quelques règles pour déterminer les types de valeurs simples et fonctionnelles:

- Regardez les littéraux
- Regardez les fonctions et autres valeurs avec lesquelles quelque chose interagit
- Regardez toutes les contraintes de type explicites
- S'il n'y a aucune contrainte, généraliser automatiquement aux types génériques

Regardez les littéraux

Le compilateur peut déduire des types en regardant les littéraux. Si le littéral est un int et que vous y ajoutez «x», alors «x» doit également être un int. Mais si le littéral est un flottant et que vous y ajoutez «x», alors «x» doit également être un flottant.

Voici quelques exemples:

```
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"
```

Regardez les fonctions et les autres valeurs avec lesquelles il interagit

S'il n'y a aucun littéral, le compilateur essaie de déterminer les types en analysant les fonctions et les autres valeurs avec lesquelles ils interagissent.

```
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
```

Examinez les contraintes de type explicites ou les annotations

Si des contraintes de type explicites ou des annotations sont spécifiées, le compilateur les utilisera.

```
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
```

Généralisation automatique

Si après tout cela, il n'y a pas de contraintes trouvées, le compilateur ne fait que rendre les types génériques.

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

Des choses qui peuvent aller mal avec l'inférence de type

L'inférence de type n'est pas parfaite, hélas. Parfois, le compilateur ne sait pas quoi faire. Encore une fois, comprendre ce qui se passe vous aidera vraiment à rester calme au lieu de vouloir tuer le compilateur. Voici quelques-unes des principales raisons des erreurs de type:

- Déclarations hors d'usage
- Pas suffisamment d'informations
- Méthodes surchargées

Déclarations hors d'usage

Une règle de base est que vous devez déclarer les fonctions avant leur utilisation.

Ce code échoue:

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

Mais c'est ok:

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

Déclarations récursives ou simultanées

Une variante du problème «hors service» se produit avec les fonctions récursives ou les définitions qui doivent se référer. Aucune réorganisation ne sera utile dans ce cas - nous devons utiliser des mots-clés supplémentaires pour aider le compilateur.

Lorsqu'une fonction est en cours de compilation, l'identifiant de la fonction n'est pas disponible pour le corps. Donc, si vous définissez une fonction récursive simple, vous obtiendrez une erreur de compilation. La solution consiste à ajouter le mot-clé «rec» dans la définition de la fonction. Par exemple:

```
// 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
```

Voici la version fixe avec «rec fib» ajouté pour indiquer qu'il est récursif:

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

Pas suffisamment d'informations

Parfois, le compilateur n'a pas assez d'informations pour déterminer un type. Dans l'exemple suivant, le compilateur ne sait pas sur quel type la méthode Length est censée fonctionner. Mais il ne peut pas non plus être générique, alors il se plaint.

```
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 ...
```

Ces types d'erreur peuvent être corrigés avec des annotations explicites.

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

Méthodes surchargées

Lorsque vous appelez une classe ou une méthode externe dans .NET, vous obtenez souvent des erreurs dues à une surcharge.

Dans de nombreux cas, comme l'exemple ci-dessous, vous devrez annoter explicitement les paramètres de la fonction externe afin que le compilateur sache quelle méthode surchargée appeler.

```
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
```

Parfois, les méthodes surchargées ont des noms d'argument différents, auquel cas vous pouvez également donner un indice au compilateur en nommant les arguments. Voici un exemple pour le constructeur StreamReader.

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

Lire Les types en ligne: https://riptutorial.com/fr/fsharp/topic/3559/les-types

Chapitre 21: Mémo

Examples

Mémo simple

La mémorisation consiste à mettre en cache les résultats des fonctions pour éviter de calculer le même résultat plusieurs fois. Ceci est utile lorsque vous travaillez avec des fonctions qui effectuent des calculs coûteux.

Nous pouvons utiliser une fonction factorielle simple comme exemple:

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

    innerLoop index 1
```

L'appel de cette fonction plusieurs fois avec le même paramètre entraîne le même calcul encore et encore.

La mémorisation nous aidera à mettre en cache le résultat factoriel et à le renvoyer si le même paramètre est transmis à nouveau.

Voici une implémentation simple de mémo:

```
// 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 fonction de `memoization` prend simplement une fonction comme paramètre et renvoie une fonction avec la même signature. Vous pouvez le voir dans la signature de la méthode `f:('a -> 'b) -> ('a -> 'b)` . De cette façon, vous pouvez utiliser la mémorisation de la même manière que

si vous appeliez la méthode factorielle.

Les appels à `printfn` doivent montrer ce qui se passe lorsque nous appelons la fonction plusieurs fois; ils peuvent être enlevés en toute sécurité.

Utiliser la mémorisation est facile:

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

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

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

Mémo dans une fonction récursive

En utilisant l'exemple précédent de calcul de la factorielle d'un entier, mettez dans la table de hachage toutes les valeurs de factorielles calculées à l'intérieur de la récursivité, qui n'apparaissent pas dans la table.

Comme dans l'article sur la mémorisation , nous déclarons une fonction `f` qui accepte un paramètre de fonction `fact` et un paramètre entier. Cette fonction `f` comprend des instructions pour calculer la factorielle de `n fact(n-1)` .

Cela permet de gérer la récursivité par la fonction retournée par `memorec` et non par la `fact` elle-même et éventuellement arrêter le calcul si le `fact(n-1)` valeur a été déjà calculé et est situé dans la table de hachage.

```
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
```

Résultat:

```
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
```

Lire Mémo en ligne: https://riptutorial.com/fr/fsharp/topic/2698/memo

Chapitre 22: Modèles actifs

Examples

Modèles actifs simples

Les modèles actifs sont un type spécial de correspondance de modèle dans lequel vous pouvez spécifier les catégories nommées dans lesquelles vos données peuvent tomber, puis utiliser ces catégories dans les instructions de `match` .

Pour définir un modèle actif qui classe les nombres comme positifs, négatifs ou nuls:

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

Cela peut ensuite être utilisé dans une expression de correspondance de motif:

```
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
```

Modèles actifs avec paramètres

Les modèles actifs ne sont que des fonctions simples.

Comme les fonctions, vous pouvez définir des paramètres supplémentaires:

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

Cela peut être utilisé dans un modèle correspondant à cette manière:

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

Les patterns actifs peuvent être utilisés pour valider et transformer des

arguments de fonction

Une utilisation intéressante mais plutôt inconnue des modèles actifs dans `F#` est qu'ils peuvent être utilisés pour valider et transformer des arguments de fonction.

Considérons la méthode classique de validation des arguments:

```
// 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 ()
```

En règle générale, nous ajoutons du code dans la méthode pour vérifier que les arguments sont corrects. En utilisant les modèles actifs dans `F#` nous pouvons généraliser ceci et déclarer l'intention dans la déclaration d'argument.

Le code suivant est équivalent au code ci-dessus:

```
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 ()
```

Pour l'utilisateur de la fonction `f` et `g` il n'y a pas de différence entre les deux versions.

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

Une préoccupation est de savoir si Active Patterns ajoute des performances supplémentaires. Utilisons `ILSpy` pour décompiler `f` et `g` pour voir si c'est le cas.

```
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();
}
```

Grâce à `inline` Active Patterns n'ajoute aucune surcharge supplémentaire par rapport à la méthode classique de validation des arguments.

Modèles actifs comme wrappers API .NET

Les patterns actifs peuvent être utilisés pour rendre l'appel de certaines API .NET plus naturel, en particulier ceux qui utilisent un paramètre de sortie pour renvoyer plus que la valeur de retour de la fonction.

Par exemple, vous appelez normalement la méthode `System.Int32.TryParse` comme suit:

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

Vous pouvez améliorer ceci un peu en utilisant la correspondance de motif:

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

Cependant, nous pouvons également définir le modèle actif suivant qui encapsule la fonction `System.Int32.TryParse` :

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

Maintenant, nous pouvons faire ce qui suit:

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

Les API d'expressions régulières sont un autre bon candidat pour être inclus dans un modèle actif:

```
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"
```

Modèles actifs complets et partiels

Il existe deux types de motifs actifs dont l'utilisation est quelque peu différente: complète et partielle.

Des modèles actifs complets peuvent être utilisés lorsque vous pouvez énumérer tous les résultats, comme "est-ce qu'un nombre impair ou pair?"

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

Notez que la définition de modèle actif répertorie à la fois les cas possibles et rien d'autre et que le corps renvoie l'un des cas répertoriés. Lorsque vous l'utilisez dans une expression de correspondance, il s'agit d'une correspondance complète:

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

Ceci est pratique lorsque vous voulez décomposer l'espace de saisie en catégories connues qui le couvrent complètement.

En revanche, les patrons actifs partiels vous permettent d'ignorer explicitement certains résultats possibles en renvoyant une `option` . Leur définition utilise un cas particulier de _ pour le cas sans correspondance.

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

De cette façon, nous pouvons égaler même si certains cas ne peuvent pas être traités par notre fonction d'analyse.

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

Les modèles actifs partiels peuvent être utilisés comme une forme de test, que l'entrée entre ou non dans une catégorie spécifique dans l'espace d'entrée, tout en ignorant les autres options.

Lire Modèles actifs en ligne: https://riptutorial.com/fr/fsharp/topic/962/modeles-actifs

Chapitre 23: Monades

Examples

Comprendre Monads vient de la pratique

Cette rubrique est destinée aux développeurs F # intermédiaires à avancés

"Que sont les Monads?" est une question commune. C'est facile à répondre, mais comme dans Hitchhikers guide to galaxy, nous réalisons que nous ne comprenons pas la réponse car nous ne savions pas ce que nous demandions après.

Beaucoup croient que la façon de comprendre Monads est de les pratiquer. En tant que programmeurs, nous ne nous soucions généralement pas des fondements mathématiques de ce que sont le principe de substitution de Liskov, ses sous-types ou sous-classes. En utilisant ces idées, nous avons acquis une intuition pour ce qu'ils représentent. La même chose est vraie pour les Monads.

Afin de vous aider à démarrer avec Monads, cet exemple montre comment créer une bibliothèque Monadic Parser Combinator . Cela pourrait vous aider à démarrer mais la compréhension viendra de l'écriture de votre propre bibliothèque Monadic.

Suffisamment de temps, temps pour le code

Le type de parseur:

```
// 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)
```

En utilisant cette définition d'un analyseur, nous définissons certaines fonctions d'analyse fondamentales

```
// 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` est une fonction qui donne une fonction `sat` un analyseur qui réussit si nous n'avons pas passé `EOS` et le caractère à la position actuelle passe la fonction `sat` . En utilisant `satisfy` nous créons un certain nombre d'analyseurs de caractères utiles.

En cours d'exécution dans 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)
```

Nous avons des analyseurs fondamentaux en place. Nous allons les combiner en analyseurs plus puissants utilisant des fonctions de combinateur d'analyseur

```
// '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)
```

Les noms et les signatures ne sont pas choisis arbitrairement mais nous ne nous en occuperons pas, voyons plutôt comment nous utilisons `bind` pour combiner un analyseur plus complexe:

```
> 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)
```

Cela nous montre que la `bind` nous permet de combiner deux analyseurs en un analyseur plus complexe. Comme résultat de `bind` est un analyseur qui à son tour peut être combiné à nouveau.

```
> 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` sera la manière fondamentale de combiner les analyseurs bien que nous définissions des fonctions d'aide pour simplifier la syntaxe.

Une des choses qui peut simplifier la syntaxe sont les expressions de calcul . Ils sont faciles à définir:

```
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)
```

Ceci est équivalent à:

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

Un autre combinateur d'analyseur fondamental que nous allons utiliser est `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
```

Cela nous permet de définir `letterOrDigit` comme ceci:

```
> 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)
```

Une note sur les opérateurs Infix

Une préoccupation commune à la PF est l'utilisation d'opérateurs infixes inhabituels tels que `>>=` , `>=>` , `<-` et ainsi de suite. Cependant, la plupart ne sont pas préoccupés par l'utilisation de `+` , `-` , `*` , `/` et `%` , ce sont des opérateurs bien connus utilisés pour composer des valeurs. Cependant, une grande partie de la planification familiale consiste à composer non seulement des valeurs, mais aussi des fonctions. Pour un développeur de FP intermédiaire, les opérateurs infixes `>>=` , `>=>` , `<-` sont bien connus et devraient avoir des signatures spécifiques ainsi que des sémantiques.

Pour les fonctions que nous avons définies jusqu'à présent, nous définirions les opérateurs infixes suivants utilisés pour combiner les analyseurs syntaxiques:

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

Donc `>>=` signifie `bind` et `<|>` signifie `orElse` .

Cela nous permet de combiner les analyseurs plus succincts:

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

Afin de définir des combinateurs d'analyse avancés qui nous permettront d'analyser des expressions plus complexes, nous définissons quelques combinateurs d'analyseurs plus 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
  }
```

Nous sommes prêts à définir `many` et plus `sepBy` car ils appliquent les analyseurs d'entrée jusqu'à ce qu'ils échouent. Puis `many` et `sepBy` renvoie le résultat agrégé:

```
// '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
```

Création d'un analyseur d'expression simple

Avec les outils que nous avons créés, nous pouvons maintenant définir un analyseur pour des expressions simples telles que `1+2*3`

Nous commençons par le bas en définissant un analyseur pour les entiers `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)
  }
```

Nous essayons d'analyser autant de chiffres que possible, le résultat est une `char list` . Si la liste est vide, nous `fail` , sinon nous plions les caractères en un entier.

Test de `pint` en FSI:

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

En outre, nous devons analyser les différents types d'opérateurs utilisés pour combiner des valeurs entières:

```
// 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)
```

Tout attacher ensemble:

```
// '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
  }
```

Tout faire en FSI:

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

Conclusion

En définissant l' `Parser<'T>` , `return_` , `bind` et en s'assurant qu'ils obéissent aux lois monadiques, nous avons construit un framework simple mais puissant de Combinator Monadic Parser.

Les monades et les analyseurs vont de pair car les analyseurs sont exécutés dans un état analyseur. Monads nous permet de combiner des analyseurs tout en masquant l'état de l'analyseur, réduisant ainsi l'encombrement et améliorant la composabilité.

Le framework que nous avons créé est lent et ne génère aucun message d'erreur, ceci afin de garder le code succinct. FParsec fournit à la fois des performances acceptables et d'excellents messages d'erreur.

Cependant, un exemple seul ne peut pas donner une compréhension de Monads. Il faut pratiquer les Monades.

Voici quelques exemples sur Monads que vous pouvez essayer d'implémenter pour atteindre vos objectifs:

1. State Monad - Autorise le transport implicite de l'état d'environnement caché
2. Tracer Monad - Permet de tracer implicitement l'état de trace. Une variante de State Monad
3. Tortue Monade - Une Monade pour créer des programmes Tortue (Logos). Une variante de State Monad
4. Continuation Monad - Une coroutine Monad. Un exemple de ceci est `async` dans F #

La meilleure chose à faire pour apprendre serait de créer une application pour Monads dans un domaine avec lequel vous êtes à l'aise. Pour moi c'était des parseurs.

Code source complet:

```
// 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
  }
```

Les expressions de calcul fournissent une syntaxe alternative à la chaîne Monads

En relation avec les monades sont des expressions de calcul F# (CE). Un programmeur implémente généralement un CE pour fournir une approche alternative à l'enchaînement des Monads, c'est-à-dire au lieu de cela:

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

Vous pouvez écrire ceci:

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

Les deux styles sont équivalents et cela dépend de la préférence du développeur.

Afin de démontrer comment implémenter un CE imaginez que toutes les traces incluent un identifiant de corrélation. Cet identifiant de corrélation aidera à corréler les traces appartenant au même appel. Ceci est très utile lorsque des fichiers journaux contiennent des traces d'appels simultanés.

Le problème est qu'il est fastidieux d'inclure l'ID de corrélation comme argument pour toutes les fonctions. Comme Monads permet de porter un état implicite, nous définirons un Log Monad pour masquer le contexte du journal (c'est-à-dire l'ID de corrélation).

Nous commençons par définir un contexte de journal et le type d'une fonction qui trace avec le contexte du journal:

```
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 ())
```

Nous définissons également deux fonctions de trace qui se connecteront avec l'identifiant de

corrélation du contexte du journal:

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

`trace` est une `Function<unit>` qui signifie qu'elle sera transmise à un contexte de journal lorsqu'elle est appelée. A partir du contexte du journal, nous prenons l'identifiant de corrélation et le trace avec `v`

De plus, nous définissons `bind` et `return_` et comme ils suivent les lois de la monade, cela forme notre Log Monad.

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

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

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

Enfin, nous définissons `LogBuilder` qui nous permettra d'utiliser la syntaxe `CE` pour chaîner `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 ()
```

Nous pouvons maintenant définir nos fonctions qui doivent avoir le contexte de journal implicite:

```
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
  }
```

Nous exécutons g avec:

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

Quelles impressions:

```
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
```

Notez que le CorrelationId est implicitement porté de `run` à `g` vers `f` ce qui nous permet de corréler les entrées du journal lors du dépannage.

`CE` a beaucoup plus de fonctionnalités mais cela devrait vous aider à définir votre propre `CE` : s.

Code complet:

```
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
```

Lire Monades en ligne: https://riptutorial.com/fr/fsharp/topic/3320/monades

Chapitre 24: Paramètres de type résolus statiquement

Syntaxe

- `s` est une instance de `^a` vous voulez accepter à la compilation, qui peut être tout ce qui implémente les membres que vous appelez réellement en utilisant la syntaxe.
- `^a` est similaire aux génériques qui seraient `'a` (ou `'A` ou `'T` par exemple) mais ceux-ci sont résolus à la compilation, et permettent tout ce qui correspond à tous les usages demandés dans la méthode. (pas d'interfaces requises)

Examples

Utilisation simple pour tout ce qui a un membre Length

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

Classe, interface, utilisation des enregistrements

```
// 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)
```

Appel de membre statique

Cela acceptera n'importe quel type avec une méthode appelée `GetLength` qui ne prend rien et retourne un int:

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

Lire Paramètres de type résolus statiquement en ligne:

https://riptutorial.com/fr/fsharp/topic/7228/parametres-de-type-resolus-statiquement

Chapitre 25: Plis

Examples

Introduction aux plis, avec quelques exemples

Les plis sont des fonctions (d'ordre supérieur) utilisées avec des séquences d'éléments. Ils réduisent `seq<'a>` en `'b` où `'b` est n'importe quel type (peut-être encore `'a`). Ceci est un peu abstrait, alors allons dans des exemples pratiques concrets.

Calcul de la somme de tous les nombres

Dans cet exemple, `'a` est un `int` . Nous avons une liste de nombres et nous voulons calculer la somme de tous les nombres. Pour résumer les numéros de la liste `[1; 2; 3]` nous écrivons

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

Permettez-moi de vous expliquer, car nous avons affaire à des listes, nous utilisons `fold` dans le module `List` , donc `List.fold` . le premier argument `fold` est une fonction binaire, le **dossier** . Le second argument est la **valeur initiale** . `fold` commence à plier la liste en appliquant consécutivement la fonction de dossier aux éléments de la liste en commençant par la valeur initiale et le premier élément. Si la liste est vide, la valeur initiale est renvoyée!

La vue d'ensemble schématique d'un exemple d'exécution ressemble à ceci:

```
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 fonction `List.sum` est approximativement `List.fold add LanguagePrimitives.GenericZero` où le zéro générique le rend compatible avec les entiers, les flottants, les grands entiers, etc.

Comptage des élémets dans une liste (`count` implémentation)

Cela se fait presque comme ci-dessus, mais en ignorant la valeur réelle de l'élément dans la liste et en ajoutant 1.

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

Cela peut aussi être fait comme ceci:

```
[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
```

Donc, vous pouvez définir `count` comme suit:

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

Trouver le maximum de liste

Cette fois, nous utiliserons `List.reduce` comme `List.fold` mais sans valeur initiale, car dans ce cas, nous ne connaissons pas le type des valeurs que nous comparons:

```
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
```

Trouver le minimum d'une liste

Tout comme pour trouver le max, le dossier est différent

```
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
```

Listes concaténantes

Ici, nous prenons la liste des listes La fonction de dossier est l'opérateur `@`

```
// [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]
```

Ou vous pouvez utiliser des opérateurs binaires comme fonction de dossier:

```
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...
```

Calcul de la factorielle d'un nombre

Même idée qu'en sommant les nombres, mais maintenant on les multiplie. si on veut la factorielle de `n` on multiplie tous les éléments de la liste `[1 .. n]` . Le code devient:

```
// 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]
```

La mise en œuvre `forall` , `exists` et `contains`

La fonction `forall` vérifie si tous les éléments d'une séquence satisfont à une condition. `exists` vérifie si au moins un élément de la liste satisfait à la condition. Nous devons d'abord savoir comment réduire une liste de valeurs `bool` . Eh bien, nous utilisons des plis de cours! Les opérateurs booléens seront nos fonctions de dossier.

Pour vérifier si tous les éléments d'une liste sont `true` nous les `&&` avec la fonction `&&` avec la valeur `true` comme valeur initiale.

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

De même, pour vérifier si un élément est `true` dans une liste booléenne, nous le réduisons avec le `||` opérateur avec `false` comme valeur initiale:

```
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
```

Retour à la `forall` et `exists` . Ici, nous prenons une liste de tout type, utilisons la condition pour transformer tous les éléments en valeurs booléennes et ensuite nous la réduisons:

```
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
```

Pour vérifier si tous les éléments dans [1; 2; 3; 4] sont plus petits que 5:

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

définir la méthode `contains` avec `exists` :

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

Ou même

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

Vérifiez maintenant si la liste [1 .. 5] contient la valeur 2:

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

Mise en œuvre `reverse` :

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

Mise en œuvre de la `map` et du `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]
```

Y a-t-il quelque chose que le `fold` ne peut pas faire? Je ne sais pas vraiment

Calcul de la somme de tous les éléments d'une liste

Pour calculer la somme des termes (de type float, int ou big entier) d'une liste de nombres, il est préférable d'utiliser List.sum. Dans d'autres cas, List.fold est la fonction la mieux adaptée pour calculer une telle somme.

1. Somme de nombres complexes

Dans cet exemple, nous déclarons une liste de nombres complexes et nous calculons la somme de tous les termes de la liste.

Au début du programme, ajoutez une référence à System.Numerics

ouvrir System.Numerics

Pour calculer la somme, nous initialisons l'accumulateur au nombre complexe 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
```

Résultat:

```
(3, 51)
```

2. Somme des nombres de type union

Supposons qu'une liste soit composée de nombres de type union (float ou int) et que vous souhaitez calculer la somme de ces nombres.

Déclarez avant le type de numéro suivant:

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

Calculez la somme des nombres de type numéro d'une liste:

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

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

Résultat:

```
13.5
```

Le premier paramètre de la fonction, qui représente l'accumulateur, est de type float et le second paramètre, qui représente un élément de la liste, est de type numéro. Mais avant d'ajouter, nous devons utiliser une correspondance de modèle et la convertir en type float quand elem est de type Int.

Lire Plis en ligne: https://riptutorial.com/fr/fsharp/topic/2250/plis

Chapitre 26: Portant C # sur F

Examples

POCO

Certains types de classes les plus simples sont les POCO.

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

Dans F # 3.0, les propriétés automatiques similaires aux propriétés automatiques C # ont été introduites,

```
// 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 création d'une instance de l'un ou l'autre est similaire,

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

Si vous pouvez utiliser des valeurs immuables, un type d'enregistrement est beaucoup plus idiomatique. F #.

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

Et ce disque peut être créé:

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

Les enregistrements peuvent également être créés sur la base d'autres enregistrements en spécifiant l'enregistrement existant et en ajoutant `with` , puis une liste de champs à remplacer:

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

Classe implémentant une interface

Les classes implémentent une interface pour respecter le contrat de l'interface. Par exemple, une classe C # peut implémenter `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();
        }
    }
}
```

Pour implémenter une interface en F #, utilisez l' `interface` dans la définition du type,

```
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 -> ()
```

Lire Portant C # sur F # en ligne: https://riptutorial.com/fr/fsharp/topic/6828/portant-c-sharp-sur-f-sharp

Chapitre 27: Processeur de boîtes aux lettres

Remarques

`MailboxProcessor` gère une file d'attente de messages interne, dans laquelle plusieurs producteurs peuvent publier des messages à l'aide de différentes variantes de la méthode `Post` . Ces messages sont ensuite récupérés et traités par un seul consommateur (sauf si vous les implémentez autrement) à l'aide des variantes `Retrieve` et `Scan` . Par défaut, la production et la consommation des messages sont sécurisées pour les threads.

Par défaut, aucune gestion des erreurs n'est fournie. Si une exception non capturée est lancée dans le corps du processeur, la fonction body se terminera, tous les messages de la file d'attente seront perdus, aucun message ne pourra plus être envoyé et le canal de réponse (si disponible) recevra une exception au lieu d'une réponse. Vous devez fournir toutes les erreurs vous-même si ce comportement ne vous convient pas.

Examples

Basic Hello World

Créons d'abord un simple "Bonjour tout le monde!" `MailboxProcessor` qui traite un type de message et imprime les messages d'accueil.

Vous aurez besoin du type de message Cela peut être n'importe quoi, mais les Unions Discriminées sont un choix naturel ici car elles listent tous les cas possibles sur un même endroit et vous pouvez facilement utiliser le filtrage lors du traitement.

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

Définissez maintenant le processeur lui-même. Cela peut être fait avec la méthode statique `MailboxProcessor<'message>.Start` qui renvoie un processeur démarré prêt à faire son travail. Vous pouvez également utiliser le constructeur, mais vous devez ensuite vous assurer de démarrer le processeur ultérieurement.

```
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 ())
```

Le paramètre à `Start` est une fonction qui fait référence au `MailboxProcessor` lui-même (qui n'existe pas encore car vous le créez, mais sera disponible une fois la fonction exécutée). Cela vous donne accès à ses différentes méthodes de `Receive` et d' `Scan` pour accéder aux messages de la boîte aux lettres. A l'intérieur de cette fonction, vous pouvez faire tout le traitement dont vous avez besoin, mais une approche habituelle est une boucle infinie qui lit les messages un par un et s'appelle après chacun.

Maintenant, le processeur est prêt, mais rien à faire! Pourquoi? Vous devez lui envoyer un message à traiter. Ceci est fait avec les variantes de la méthode `Post` - utilisons le plus simple, celui du feu et de l'oubli.

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

Cela place un message dans la file d'attente interne du `processor` , la boîte aux lettres, et retourne immédiatement afin que le code appelant puisse continuer. Une fois que le processeur aura récupéré le message, il le traitera, mais cela sera fait de manière asynchrone lors de son envoi, et cela se fera probablement sur un thread séparé.

Peu après, vous devriez voir le message `"Hi, Alice! This is mailbox processor's inner loop!"` imprimé à la sortie et vous êtes prêt pour des échantillons plus compliqués.

Mutable State Management

Les processeurs de boîtes aux lettres peuvent être utilisés pour gérer l'état mutable de manière transparente et sécurisée. Construisons un compteur 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
```

Maintenant, générons des opérations

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

Et vous verrez le journal suivant

```
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
```

Concurrence

Comme le processeur de boîte aux lettres traite les messages un par un et qu'il n'y a pas d'entrelacement, vous pouvez également produire les messages à partir de plusieurs threads et vous ne verrez pas les problèmes typiques des opérations perdues ou dupliquées. Un message ne permet en aucun cas d'utiliser l'ancien état des autres messages, à moins que vous ne l'implémentiez spécifiquement.

```
let processor = createProcessor 0

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

Tous les messages sont publiés à partir de différents threads. L'ordre dans lequel les messages sont publiés dans la boîte aux lettres n'est pas déterministe. Par conséquent, l'ordre de traitement n'est pas déterministe, mais comme le nombre total d'incréments et de décréments est équilibré, l'état final sera 0, peu importe dans quel ordre et à partir de quels threads les messages ont été envoyés.

Véritable état mutable

Dans l'exemple précédent, nous n'avons simulé qu'un état mutable en transmettant le paramètre

de boucle récursif, mais le processeur de boîte aux lettres possède toutes ces propriétés même pour un état véritablement mutable. Ceci est important lorsque vous maintenez un état élevé et que l'immutabilité est impraticable pour des raisons de performance.

Nous pouvons réécrire notre compteur à l'implémentation suivante

```
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 ())
```

Même si cela ne serait certainement pas thread-safe si l'état du compteur était modifié directement à partir de plusieurs threads, vous pouvez voir en utilisant le message parallèle Posts from previous section que le processeur de boîte aux lettres traite les messages les uns après les autres sans entrelacement. valeur la plus courante.

Valeurs de retour

Vous pouvez retourner une valeur asynchrone pour chaque message traité si vous envoyez un `AsyncReplyChannel<'a>` dans le message.

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

Ensuite, le processeur de boîte aux lettres peut utiliser ce canal lors du traitement du message pour renvoyer une valeur à l'appelant.

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

Maintenant, pour créer un message, vous avez besoin d' `AsyncReplyChannel<'a>` - qu'est-ce que c'est et comment créez-vous une instance de travail? Le meilleur moyen est de laisser MailboxProcessor le fournir pour vous et d'extraire la réponse à un `Async<'a>` plus commun

`Async<'a>` . Cela peut être fait en utilisant par exemple la méthode `PostAndAsynReply` , où vous ne publiez pas le message complet, mais plutôt une fonction de type (dans notre cas) `AsyncReplyChannel<OutputData> -> MessageWithResponse` :

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

Cela affichera le message dans une file d'attente et attendra la réponse, qui arrivera une fois que le processeur aura atteint ce message et aura répondu en utilisant le canal.

Il existe également une variante synchrone `PostAndReply` qui bloque le thread appelant jusqu'à ce que le processeur réponde.

Traitement des messages hors service

Vous pouvez utiliser les méthodes `Scan` ou `TryScan` pour rechercher des messages spécifiques dans la file d'attente et les traiter, quel que soit le nombre de messages devant eux. Les deux méthodes examinent les messages dans la file d'attente dans l'ordre où elles sont arrivées et recherchent un message spécifié (jusqu'à la temporisation facultative). Au cas où il n'y aurait pas de message, `TryScan` renverrait None, tandis que `Scan` continuerait d'attendre que ce message arrive ou que l'opération arrive à `TryScan` .

Voyons cela en pratique. Nous voulons que le processeur traite `RegularOperations` quand c'est possible, mais chaque fois qu'il y a un `PriorityOperation` , il doit être traité dès que possible, peu importe le nombre d'autres `RegularOperations` dans la file d'attente.

```
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())
```

Lire Processeur de boîtes aux lettres en ligne:
https://riptutorial.com/fr/fsharp/topic/9409/processeur-de-boites-aux-lettres

Chapitre 28: Réflexion

Examples

Réflexion robuste à l'aide de citations F

La réflexion est utile mais fragile. Considère ceci:

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

Les problèmes avec ce type de code sont les suivants:

1. Le code ne fonctionne pas car il y a plusieurs surcharges de `String.StartsWith`
2. Même s'il n'y avait pas de surcharge, les versions ultérieures de la bibliothèque pourraient ajouter une surcharge provoquant un plantage à l'exécution
3. Les outils de refactoring comme les `Rename methods` sont rompus avec la réflexion.

Cela signifie que nous obtenons un crash à l'exécution pour quelque chose qui est connu à la compilation. Cela semble sous-optimal.

En utilisant les citations `F#` il est possible d'éviter tous les problèmes ci-dessus. Nous définissons des fonctions d'assistance:

```
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
```

Nous utilisons les fonctions comme ceci:

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

Cela imprime:

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

`<@ ... @>` signifie qu'au lieu d'exécuter l'expression à l'intérieur de `F#` génère un arbre d'expression représentant l'expression. `<@ "".StartsWith "" @>` génère un arbre d'expression qui ressemble à ceci: `Call (Some (Value ("")), StartsWith, [Value ("")])` . Cet arbre d'expression correspond à ce que `getMethodInfo` attend et renvoie les informations de méthode correctes.

Cela résout tous les problèmes énumérés ci-dessus.

Lire Réflexion en ligne: https://riptutorial.com/fr/fsharp/topic/4124/reflexion

Chapitre 29: Séquence

Examples

Générer des séquences

Il existe plusieurs façons de créer une séquence.

Vous pouvez utiliser les fonctions du module 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
```

Vous pouvez également utiliser l'expression de séquence:

```
// 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 }
}
```

Introduction aux séquences

Une séquence est une série d'éléments pouvant être énumérés. C'est un alias de System.Collections.Generic.IEnumerable et lazy. Il stocke une série d'éléments du même type

(peut être n'importe quelle valeur ou objet, même une autre séquence). Les fonctions du Seq.module peuvent être utilisées pour y opérer.

Voici un exemple simple d'énumération de séquence:

```
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
```

Sortie:

```
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; ...]
```

Appliquer une fonction à chaque élément d'une séquence en utilisant Seq.map

Seq.filter

Supposons que nous ayons une suite d'entiers et que nous voulons créer une séquence contenant uniquement les entiers pairs. Nous pouvons obtenir ce dernier en utilisant la fonction de `filter` du module Seq. La fonction de `filter` a le type signature `('a -> bool) -> seq<'a> -> seq<'a>` ; cela indique qu'il accepte une fonction qui retourne true ou false (parfois appelé prédicat) pour une entrée donnée de type `'a` et une séquence qui comprend des valeurs de type `'a` pour donner une séquence comprenant des valeurs de type `'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
```

Séquences répétées infinies

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

Des séquences répétées peuvent être créées en utilisant une expression de calcul `seq {}`

Lire Séquence en ligne: https://riptutorial.com/fr/fsharp/topic/2354/sequence

Chapitre 30: Syndicats discriminés

Examples

Nommer des éléments de tuples dans des unions discriminées

Lorsque vous définissez des unions discriminées, vous pouvez nommer des éléments de types de tuple et utiliser ces noms lors de la correspondance de modèle.

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

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

En outre, la dénomination des éléments des unions discriminées améliore la lisibilité du code et l 'interopérabilité avec les noms fournis par C # sera utilisée pour les noms de propriétés et les paramètres des constructeurs. Les noms générés par défaut dans le code interop sont "Item", "Item1", "Item2" ...

Usage de base discriminant des syndicats

Les unions discriminées dans F # permettent de définir des types pouvant contenir un nombre quelconque de types de données différents. Leur fonctionnalité est similaire à celle des unions C ++ ou des variantes VB, mais avec l'avantage supplémentaire d'être sans risque de type.

```
// 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
```

Union-style unions

Il n'est pas nécessaire d'inclure les informations de type dans les cas d'un syndicat discriminé. En omettant les informations de type, vous pouvez créer une union qui représente simplement un

ensemble de choix, similaire à un enum.

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

Conversion vers et depuis des chaînes avec Reflection

Parfois, il est nécessaire de convertir une union discriminée à partir d'une chaîne:

```
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
```

Union discriminée dans un seul cas

Un seul syndicat discriminé est comme n'importe quel autre syndicat discriminé, sauf qu'il n'a qu'un seul cas.

```
// 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
```

Il est utile pour renforcer la sécurité des types et couramment utilisé dans F #, par opposition à C # et Java, où la création de nouveaux types est associée à des frais supplémentaires.

Les deux définitions de type suivantes permettent de déclarer la même union discriminée:

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

Utiliser des syndicats discriminés à cas unique comme enregistrements

Parfois, il est utile de créer des types d'union avec un seul cas pour implémenter des types de type enregistrement:

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)

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

Celles-ci deviennent très utiles car elles peuvent être décomposées via une correspondance de modèle de la même manière que les arguments tuple peuvent:

```
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

Avec l'attribut `RequireQualifiedAccess` , les cas d'union doivent être appelés `MyUnion.MyCase` au lieu de `MyCase` . Cela empêche les collisions de noms dans l'espace de noms ou le module englobant:

```
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, par exemple, le `System` a été ouvert, `Single` fait référence à `System.Single` . Il n'y a pas de collision avec le cas du syndicat `Requirements.Single` .

Unions discriminatoires récursives

Type récursif

Les syndicats discriminés peuvent être récursifs, c'est-à-dire qu'ils peuvent se référer à eux-mêmes dans leur définition. Le principal exemple est un arbre:

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

A titre d'exemple, définissons l'arbre suivant:

```
    1
  2   5
3   4
```

Nous pouvons définir cet arbre en utilisant notre union discriminée récursive comme suit:

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

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

Itérer sur l'arbre n'est plus qu'une question de correspondance:

```
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]
```

Types récursifs mutuellement dépendants

Une manière d'obtenir la récursivité consiste à avoir des types imbriqués dépendants les uns des autres.

```
// 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 définition d'un type d'enregistrement directement à l'intérieur d'une union discriminée est obsolète:

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

Vous pouvez utiliser le mot `and` clé `and` pour enchaîner les définitions mutuellement dépendantes:

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

```
| ArithmeticExpr of Arithmetic
```

Lire Syndicats discriminés en ligne: https://riptutorial.com/fr/fsharp/topic/1025/syndicats-discrimines

Chapitre 31: Types d'option

Examples

Définition de l'option

Une `Option` est une union discriminée avec deux cas, `None` ou `Some` .

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

Utilisez Option <'T> sur des valeurs nulles

Dans les langages de programmation fonctionnels comme `null` valeurs `null F#` sont considérés comme potentiellement nuisibles et de style médiocre (non idiomatique).

Considérez ce code `C#` :

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

`x.Length` lancera si `x` est `null` ajoutons une protection:

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

Ou:

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

Ou:

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

Dans les idiomatiques `F# null` valeurs `null` ne sont pas utilisées, notre code ressemble à ceci:

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

Cependant, il est parfois nécessaire de représenter des valeurs vides ou invalides. Ensuite, nous pouvons utiliser `Option<'T>` :

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

`SomeFunction` soit retourne `Some string` valeur ou `None` . Nous extrayons la valeur de `string` utilisant

une correspondance de modèle

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

La raison pour laquelle ce code est moins fragile que:

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

C'est parce que nous ne pouvons pas appeler `Length` sur une `string option` . Nous devons extraire la valeur de `string` utilisant la correspondance de modèle et, ce faisant, nous sommes sûrs que la valeur de `string` est sûre à utiliser.

Le module d'option permet la programmation orientée chemin de fer

La gestion des erreurs est importante mais peut rendre un algorithme élégant en désordre. La programmation orientée chemin de fer (`ROP`) est utilisée pour rendre la gestion des erreurs élégante et composable.

Considérons la fonction 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
```

Le but de `f` est d'analyser l'entrée `string` valeur (s'il y a `Some`) en un `int` . Si l' `int` est supérieur à `0` on le transforme en `float` . Dans tous les autres cas, nous renflouons avec `None` .

Bien que la fonction imbriquée soit extrêmement simple, la `match` imbriquée diminue considérablement la lisibilité.

`ROP` observe que nous avons deux types de chemins d'exécution dans notre programme

1. Happy path - calculera éventuellement une `Some` valeur
2. Chemin d'erreur - Tous les autres chemins produisent `None`

Comme les chemins d’erreur sont plus fréquents, ils ont tendance à prendre le relais. Nous aimerions que le code de chemin heureux soit le chemin de code le plus visible.

Une fonction équivalente `g` utilisant `ROP` pourrait ressembler à ceci:

```
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
```

Cela ressemble beaucoup à la façon dont nous avons tendance à traiter les listes et les séquences en `F#` .

On peut voir une `Option<'T>` comme une `List<'T>` qui ne peut contenir que `0` ou `1` élément où `Option.bind` se comporte comme `List.pick` (conceptuellement, `Option.bind` mieux `List.collect` à `List.collect` mais `List.pick` peut être plus facile à comprendre).

`bind` , `filter` et `map` gère les chemins d'erreur et `g` ne contient que le code de chemin heureux.

Toutes les fonctions qui acceptent directement l' `Option<_>` et renvoient l' `Option<_>` sont directement composables avec `|>` et `>>` .

`ROP` augmente donc la lisibilité et la composabilité.

Utiliser les types d'option de C

Ce n'est pas une bonne idée d'exposer les types d'options au code C #, car C # ne permet pas de les gérer. Les options sont soit d'introduire `FSharp.Core` comme une dépendance dans votre projet C # (ce que vous devez faire si vous utilisez une bibliothèque F # *non* conçue pour l'interopérabilité avec C #), soit de changer les valeurs `None` en `null` .

Pré-F # 4.0

Pour ce faire, créez votre propre fonction de conversion:

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

Pour les types de valeur, vous devez avoir recours à la boxe ou à `System.Nullable` .

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

F # 4.0

Dans F # 4.0, les fonctions `ofObj` , `toObj` , `ofNullable` et `toNullable` introduites dans le module `Option` . Dans F # interactive, ils peuvent être utilisés comme suit:

```
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
```

Notez que `None` compile en `null` interne. Cependant, en ce qui concerne F #, il s'agit d'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
```

Lire Types d'option en ligne: https://riptutorial.com/fr/fsharp/topic/3175/types-d-option

Chapitre 32: Unités de mesure

Remarques

Unités au runtime

Les unités de mesure sont utilisées uniquement pour la vérification statique par le compilateur et ne sont pas disponibles au moment de l'exécution. Ils ne peuvent pas être utilisés en réflexion ou dans des méthodes comme `ToString` .

Par exemple, C # donne un `double` sans unités pour un champ de type `float<m>` défini et exposé depuis une bibliothèque F #.

Examples

Assurer la cohérence des unités dans les calculs

Les unités de mesure sont des annotations de type supplémentaires pouvant être ajoutées à des flottants ou à des nombres entiers. Ils peuvent être utilisés pour vérifier lors de la compilation que les calculs utilisent des unités de manière cohérente.

Pour définir des annotations:

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

Une fois définies, les annotations peuvent être utilisées pour vérifier qu'une expression génère le type attendu.

```
// 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
```

Conversions entre unités

```
[<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>
```

Notez que le compilateur F # ne sait pas que `1<m>` est égal à `100<cm>` . En ce qui concerne les soins, les unités sont des types distincts. Vous pouvez écrire des fonctions similaires pour convertir des mètres en kilogrammes et le compilateur ne s'en souciera pas.

```
[<Measure>] type kg

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

Il n'est pas possible de définir des unités de mesure sous la forme de multiples d'autres unités telles que

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

Cependant, il est très simple de définir des unités "par quelque chose", par exemple Hertz, la fréquence de mesure est simplement "par seconde".

```
// 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
```

Utiliser LanguagePrimitives pour préserver ou définir des unités

Lorsqu'une fonction ne préserve pas automatiquement les unités en raison d'opérations de niveau inférieur, le module `LanguagePrimitives` peut être utilisé pour définir des unités sur les primitives qui les prennent en charge:

```
/// This cast preserves units, while changing the underlying type
```

```
let inline castDoubleToSingle (x : float<'u>) : float32<'u> =
    LanguagePrimitives.Float32WithMeasure (float32 x)
```

Pour affecter des unités de mesure à une valeur à virgule flottante double précision, multipliez simplement par une unité avec les unités correctes:

```
[<Measure>]
type USD

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

Pour affecter des unités de mesure à une valeur sans unité qui n'est pas System.Double, par exemple, provenant d'une bibliothèque écrite dans une autre langue, utilisez une conversion:

```
open LanguagePrimitives

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

Voici les types de fonctions signalés par F # interactive:

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

Paramètres de type d'unité de mesure

L'attribut `[<Measure>]` peut être utilisé sur les paramètres de type pour déclarer les types génériques par rapport aux unités de mesure:

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

Test d'utilisation:

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

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

Utiliser des types d'unité standardisés pour maintenir la compatibilité

Par exemple, les types pour les unités SI ont été normalisés dans la bibliothèque principale F #, dans `Microsoft.FSharp.Data.UnitSystems.SI` . Ouvrez le sous-espace de noms approprié, `UnitNames` ou `UnitSymbols` , pour les utiliser. Ou, si seulement quelques unités SI sont requises, elles peuvent être importées avec des alias de type:

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

Certains utilisateurs ont tendance à faire ce qui suit, ce qui **ne devrait pas être fait** chaque fois qu'une définition est déjà disponible:

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

La différence devient évidente lors de l'interfaçage avec un autre code faisant référence aux types de SI standard. Le code faisant référence aux unités standard est compatible, tandis que le code qui définit son propre type est incompatible avec tout code n'utilisant pas sa définition spécifique.

Par conséquent, utilisez toujours les types standard pour les unités SI. Peu importe que vous fassiez référence à `UnitNames` ou `UnitSymbols` , puisque les noms équivalents dans ces deux `UnitSymbols` font référence au même type:

```
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>
```

Lire Unités de mesure en ligne: https://riptutorial.com/fr/fsharp/topic/1055/unites-de-mesure

Chapitre 33: Workflows de séquence

Examples

rendement et rendement!

Dans les flux de travail séquentiels, le `yield` ajoute un seul élément dans la séquence en cours de création. (Dans la terminologie monadique, c'est le `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!` (prononcé *bang de rendement*) insère tous les éléments d'une autre séquence dans cette séquence en cours de construction. Ou, en d'autres termes, il ajoute une séquence. (En ce qui concerne monades, on `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']
```

Les séquences créées par les flux de travail de séquence sont également paresseuses, ce qui signifie que les éléments de la séquence ne sont pas réellement évalués tant qu'ils ne sont pas nécessaires. Quelques méthodes pour forcer les éléments incluent l'appel `Seq.take` (extrait les n premiers éléments dans une séquence), `Seq.iter` (applique une fonction à chaque élément pour exécuter les effets secondaires) ou `Seq.toList` (convertit une séquence en liste) . Combiner cela avec la récursivité est là où le `yield!` commence vraiment à briller.

```
> 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]
```

pour

`for` séquence, l'expression est conçue pour ressembler à son cousin le plus célèbre, l'impératif for-loop. Il "boucle" à travers une séquence et évalue le corps de chaque itération dans la séquence qu'il génère. Tout comme toute séquence liée, elle n'est PAS 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]
```

Lire Workflows de séquence en ligne: https://riptutorial.com/fr/fsharp/topic/2785/workflows-de-sequence

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec F #	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	1: F # Code WPF derrière l'application avec FsXaml	Bent Tranberg
3	Cordes	FireAlkazar, Julien Pires
4	Correspondance de motif	asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene
5	Des classes	asibahi, inzi, RamenChef, Tomasz Maczyński
6	Des dossiers	eirik, goric, Ringil
7	Des listes	asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj
8	En utilisant F #, WPF, FsXaml, un menu et une boîte de dialogue	Bob McCrory, Goswin
9	Évaluation paresseuse	inzi
10	Extensions de type et de module	Jono Job
11	F # Performance Tips and Tricks	FuleSnabel, Paul Westcott, Ringil, s952163
12	F # sur .NET Core	Boggin, Joel Martinez
13	Fournisseurs de type	GregC, jdphenix, Joel Martinez
14	Génériques	Jake Lishman
15	Implémentation d'un modèle de conception dans F #	FuleSnabel, Ringil

16	Introduction à WPF en F #	Funk
17	Le type "unit"	4444, Abel, Jake Lishman, rmunn
18	Les fonctions	asibahi, Julien Pires, rmunn, ronilk
19	Les opérateurs	FuleSnabel
20	Les types	Cedric Royer-Bertrand, FuleSnabel, Julien Pires
21	Mémo	Jean-Claude Colette, Julien Pires, Ringil
22	Modèles actifs	Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
23	Monades	FuleSnabel
24	Paramètres de type résolus statiquement	Maslow
25	Plis	Jean-Claude Colette, Zaid Ajaj
26	Portant C # sur F #	jdphenix, marklam, RamenChef
27	Processeur de boîtes aux lettres	Honza Brestan
28	Réflexion	FuleSnabel
29	Séquence	Foggy Finder, inzi, James McCalden, Julien Pires, s952163
30	Syndicats discriminés	chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen, Vandroiy, VillasV
31	Types d'option	asibahi, chillitom, FuleSnabel
32	Unités de mesure	asibahi, goric, GregC, Vandroiy
33	Workflows de séquence	Jwosty