Бесплатная электронная книга

УЧУСЬ F#

Free unaffiliated eBook created from
Stack Overflow contributors.

#f#

...... 1

1: F # 2

...... 2

...... 2

Examples 2

...... 2

Windows 2

OS X 2

Linux 3

, ! 3

F # 3

2: 1: F # WPF FsXaml 5

...... 5

Examples 5

WPF F # 5

3: 7

4: 7

2: 8

...... 9

3: F # .NET Core 11

Examples 11

dotnet CLI 11

...... 11

4: F # 12

Examples 12

...... 12

...... 13

F # 22

5: 32

Examples 32

...... 32

...... 32

...... 33

.NET API 34

...... 35

6: WPF F # 37

...... 37

...... 37

Examples 37

FSharp.ViewModule 37

Gjallarhorn 39

7: 43

Examples 43

...... 43

...... 43

8: 45

Examples 45

...... 45

...... 45

enum 45

...... 46

...... 46

...... 46

RequireQualifiedAccess 47

...... 47

...... 47

...... 48

9: 50

Examples 50

- 50

...... 50

10: 51

...... 51

.... 51

Examples 51

.... 51

.... 51

LanguagePrimitives 52

.... 53

.... 53

11: F #, WPF, FsXaml, ... 55

.... 55

Examples 55

.... 55

«-» 55

XAML 57

XAML F # 58

MainWindow.xaml 62

App.xaml App.xaml.fs, ... 63

12: 65

Examples 65

.... 65

.... 65

13: 66

Examples 66

.... 66

F # 66

14: 68

Examples 68

.... 68

.... 69

15: 71

Examples 71

.... 71

Monads 79

16: **83**

Examples 83

.... 83

Latebinding F # ? 84

17: **86**

Examples 86

F # 86

18: **88**

.... 88

Examples 88

, ... 88

, , ... 88

.... 88

19: C # F # **90**

Examples 90

POCO 90

.... 91

20: **92**

Examples 92

.... 92

.... 92

Seq.map 93

Seq.filter 93

.... 93

21: **95**

Examples 95

! 95

.... 96

22: **97**

.... 97

Examples 97

Hello World 97

....98
....99
....100
....100
....101
23: **103**
....103
Examples....103
/ 103
....104
....104
24: F #....**106**
Examples....106
, F #....106
25: **109**
Examples....109
, ...109
....**109**
(count)....**109**
....**110**
....**110**
....**110**
....**111**
forall exists contains....**111**
reverse :....**112**
map filter....**112**
....112
26: **115**
....115
Examples....115
....115

.......115

.......**115**

bools , ints **115**

_ **116**

.......116

.......117

27: **118**

.......118

Examples.......118

.......118

.......118

.......119

28: **122**

Examples.......122

.......122

.......122

29: «».......**124**

Examples.......124

0-?.......124

.......125

30: **127**

Examples.......127

CSV-.......127

WMI.......127

31: **128**

Examples.......128

.......128

.......128

F #, ...129

.......131

32: **135**

Examples 135

135

<'T> 135

136

C # 137

Pre-F # 4.0 **137**

F # 4.0 **137**

33: **139**

Examples 139

139

140

Curried vs Tupled Functions 141

141

142

144

Около

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#.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

глава 1: Начало работы с F

замечания

F # является «функционально-первым» языком. Вы можете узнать обо всех различных типах выражений вместе с функциями .

Компилятор F #, который является открытым исходным кодом, компилирует ваши программы в IL, что означает, что вы можете использовать код F # с любого совместимого с .NET языка, такого как C # ; и запустить его на Mono, .NET Core или .NET Framework в Windows.

Версии

Версия	Дата выхода
1.x	2005-05-01
2,0	2010-04-01
3.0	2012-08-01
3,1	2013-10-01
4,0	2015-07-01

Examples

Установка или настройка

Windows

Если у вас установлена Visual Studio (любая версия, включая экспресс и сообщество), F_ уже должен быть включен. Просто выберите F # в качестве языка при создании нового проекта. Или просмотрите http://fsharp.org/use/windows/ для получения дополнительных параметров.

OS X

Xamarin Studio поддерживает F #. В качестве альтернативы вы можете использовать VS

Code для OS X , который является кросс-платформенным редактором Microsoft. Сделав установку VS-кода, запустите `VS Code Quick Open` (`Ctrl` + `P`), затем запустите `ext install Ionide-fsharp`

Вы также можете рассмотреть Visual Studio для Mac .

Существуют и другие альтернативы, описанные здесь .

Linux

Установите пакеты с `mono-complete` и `fsharp` через `fsharp` пакетов вашего дистрибутива (Apt, Yum и т. Д.). Для хорошего редактирования используйте либо код Visual Studio , либо установите плагин `ionide-fsharp` , либо используйте Atom и установите плагин `ionide-installer` . Дополнительную информацию см. В http://fsharp.org/use/linux/ .

Привет, мир!

Это код для простого консольного проекта, который печатает «Hello, World!». на STDOUT и выйдет с кодом выхода `0`

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

Пример разбивки Поэтапный:

- `[<EntryPoint>]` - Атрибут .net, который отмечает «метод, который вы используете для установки точки входа» вашей программы (источник).
- `let main argv` - это определяет функцию `main` с одним параметром `argv` . Поскольку это точка входа в программу, `argv` будет массивом строк. Содержимое массива - это аргументы, которые были переданы программе, когда они были выполнены.
- `printfn "Hello, World!"` - функция `printfn` выводит строку **, переданную в качестве первого аргумента, также добавляя новую строку.
- `0` - Функции F # всегда возвращают значение, а возвращаемое значение является результатом последнего выражения в функции. Помещение `0` в качестве последней строки означает, что функция всегда будет возвращать ноль (целое число).

** На самом деле это *не* строка, хотя она похожа на одну. Это на самом деле TextWriterFormat , который по выбору позволяет использовать статические типы проверенных аргументов. Но для примера «hello world» его можно представить как строку.

F # Интерактивный

F # Interactive, это среда REPL, которая позволяет выполнять код F #, по одной строке за

раз.

Если вы установили Visual Studio с F #, вы можете запустить F # Interactive в консоли, набрав `"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"` . В Linux или OS X вместо этого команда `fsharpi` , которая должна быть либо в `/usr/bin` либо в `/usr/local/bin` зависимости от того, как вы установили F # - в любом случае, команда должна быть на вашем `PATH` чтобы вы могли просто введите `fsharpi` .

Пример интерактивного использования 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
```

Использовать `#help;;` за помощью

Обратите внимание на использование `;;` чтобы сообщить REPL о выполнении любых ранее введенных команд.

Прочитайте Начало работы с F # онлайн: https://riptutorial.com/ru/fsharp/topic/817/начало-работы-с-f-sharp

глава 2: 1: F # Код WPF для приложения с FsXaml

Вступление

Большинство примеров, найденных для программирования F # WPF, похоже, имеют отношение к шаблону MVVM, а некоторые из них - MVC, но рядом нет ни одного, который правильно показывает, как встать и работать с «старым старым» кодом.

Код за рисунком очень прост в использовании для обучения, а также экспериментов. Он используется в многочисленных вводных книгах и учебных материалах в Интернете. Вот почему.

Эти примеры продемонстрируют, как создать код за приложением с окнами, элементами управления, изображениями и значками и т. Д.

Examples

Создайте новый код WPF F # для приложения.

Создайте консольное приложение F #.

Измените **тип вывода** приложения на приложение *Windows* .

Добавьте пакет **FsXaml** NuGet.

Добавьте эти четыре исходных файла в порядке, указанном здесь.

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.
```

Удалите файл *Program.fs* из проекта.

Измените действие **сборки** на *ресурс* для двух файлов xaml.

Добавьте ссылку на сборку **UIAutomationTypes** сборки .NET.

Скомпилируйте и запустите.

Вы не можете использовать конструктор для добавления обработчиков событий, но это совсем не проблема. Просто добавьте их вручную в код позади, как вы видите с тремя

обработчиками в этом примере, включая обработчик тестовой кнопки.

UPDATE: в FsXaml добавлен альтернативный и, возможно, более элегантный способ добавления обработчиков событий. Вы можете добавить обработчик событий в XAML, как и в C #, но вы должны сделать это вручную, а затем переопределить соответствующий член, который появляется в вашем F #. Я рекомендую это.

3: добавление значка в окно

Рекомендуется сохранять все значки и изображения в одной или нескольких папках.

Щелкните правой кнопкой мыши по проекту и используйте F # Power Tools / New Folder для создания папки с именем «Изображения».

На диске поместите значок в папку «Новые *изображения* ».

Вернитесь в Visual Studio, щелкните правой кнопкой мыши по *изображению* и используйте « **Добавить** / **существующий элемент»** , а затем отобразите « *Все файлы» (.*) **, чтобы увидеть файл значка, чтобы его можно было выбрать, а затем « **Добавить»** .

Выберите файл значка и установите его **действие «Построение»** в « *Ресурс»* .

В MainWindow.xaml используйте атрибут Icon как это. Окружающие линии показаны для контекста.

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

Перед тем, как запустить, выполните перестройку, а не просто сборку. Это связано с тем, что Visual Studio не всегда помещает файл значка в исполняемый файл, если вы не перестраиваете его.

Это окно, а не приложение, которое теперь имеет значок. Вы увидите значок в левом верхнем углу окна во время выполнения, и вы увидите его на панели задач. Диспетчер задач и Проводник Windows не будут показывать этот значок, потому что они отображают значок приложения, а не значок окна.

4: Добавить значок в приложение

Создайте текстовый файл с именем AppIcon.rc со следующим содержимым.

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

Для этого вам понадобится файл значков с именем AppIcon.ico, но, конечно, вы можете настроить имена по своему вкусу.

Выполните следующую команду.

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

Если вы не можете найти rc.exe в этом месте, найдите его ниже **C: \ Program Files (x86) \ Windows Kits** . Если вы все еще не можете найти его, загрузите Windows SDK из Microsoft.

Будет создан файл с именем AppIcon.res.

В Visual Studio откройте свойства проекта. Выберите страницу **приложения** .

В текстовом поле « **Файл ресурсов»** введите *AppIcon.res* (или *Images \ AppIcon.res,* если вы положили его там), а затем закройте свойства проекта для сохранения.

Появится сообщение об ошибке: «Введенный файл ресурсов не существует. Игнорируйте это. Сообщение об ошибке не появится снова.

Перестроить. Затем исполняемый файл имеет значок приложения, и это отображается в Проводнике. При запуске этот значок также появится в диспетчере задач.

2: Добавить элемент управления

Добавьте эти два файла в этом порядке над файлами для главного окна.

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()
```

Для **Action** для *MyControl.xaml* должен быть установлен *ресурс* .

Вы, конечно, позже должны добавить «как это» в объявление MyControl, как это сделано

для главного окна.

В файле **MainWindow.xaml.fs** в классе для MainWindow добавьте эту строку

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

и добавьте эти две строки в **do-** section основного класса окна.

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

Там может быть более чем один **сделать** -сече- в классе, и вы, вероятно, понадобится при написании много кода за кодом.

Элемент управления получил светло-зеленый цвет фона, так что вы можете легко увидеть, где он находится.

Имейте в виду, что элемент управления заблокирует кнопку главного окна из представления. Это не входит в объем этих примеров, чтобы научить вас WPF в целом, поэтому мы не будем это исправлять здесь.

Как добавить элементы управления из сторонних библиотек

Если вы добавите элементы управления из сторонних библиотек в проект C # WPF, в файле XAML обычно будут такие строки, как этот.

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

Возможно, это не с FsXaml.

Дизайнер и компилятор принимают эту строку, но, скорее всего, во время выполнения, вероятно, будет исключение, жалуясь на тип третьей стороны, который не будет найден при чтении XAML.

Вместо этого попробуйте следующее.

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

Тогда это пример элемента управления, который зависит от вышесказанного.

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

Библиотека, используемая в этом примере, - это расширенный набор инструментов Wpf, доступный бесплатно через NuGet или установщик. Если вы загружаете библиотеки через NuGet, элементы управления недоступны в панели инструментов, но они все еще

отображаются в дизайнере, если вы добавите их вручную в XAML, а свойства доступны на панели «Свойства».

Прочитайте 1: F # Код WPF для приложения с FsXaml онлайн: https://riptutorial.com/ru/fsharp/topic/9008/1--f-sharp-код-wpf-для-приложения-с-fsxaml

глава 3: F # на .NET Core

Examples

Создание нового проекта через dotnet CLI

После того, как вы установили инструменты .NET CLI, вы можете создать новый проект с помощью следующей команды:

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

Это создает программу командной строки.

Начальный рабочий процесс проекта

Создать новый проект

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

Восстановите все пакеты, перечисленные в файле project.json

```
dotnet restore
```

Файл project.lock.json должен быть выписан.

Выполнить программу

```
dotnet run
```

Вышеприведённый код будет скомпилирован, если потребуется.

Результат проекта по умолчанию, созданного `dotnet new -lf#` содержит следующее:

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

Прочитайте F # на .NET Core онлайн: https://riptutorial.com/ru/fsharp/topic/4404/f-sharp-на--net-core

глава 4: F # Советы и подсказки производительности

Examples

Использование хвостовой рекурсии для эффективной итерации

Исходя из императивных языков, многие разработчики задаются вопросом, как написать `for-loop` который выходит раньше, поскольку `F#` не поддерживает `break` , `continue` или `return` . Ответ в `F#` заключается в использовании хвостовой рекурсии, которая является гибким и идиоматическим способом итерации, при этом обеспечивая отличную производительность.

Предположим, мы хотим реализовать `tryFind` для `List` . Если `F#` поддерживает `return` мы бы немного напишем `tryFind` следующим образом:

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

Это не работает в `F#` . Вместо этого мы записываем функцию с помощью tail-recursion:

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

Хвост-рекурсия выполняется в `F#` потому что, когда компилятор `F#` обнаруживает, что функция является хвостовой рекурсивной, она переписывает рекурсию в эффективный `while-loop` . Используя `ILSpy` мы видим, что это верно для нашего `loop` функций:

```
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;
}
```

Помимо некоторых ненужных назначений (которые, надеюсь, удаляет JIT-er), это, по сути, эффективный цикл.

Кроме того, хвостовая рекурсия является идиоматической для `F#` поскольку она позволяет избежать изменчивого состояния. Рассмотрим `sum` функцию , которая суммирует все элементы в `List` . Очевидная первая попытка:

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

Если мы перепишем цикл в хвостовую рекурсию, мы можем избежать изменчивого состояния:

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

Для эффективности компилятор `F#` преобразует это в `while-loop` который использует изменчивое состояние.

Измерьте и проверьте свои предположения о производительности

Этот пример написан с учетом `F#` но идеи применимы во всех средах

Первое правило при оптимизации производительности - не полагаться на предположение; Всегда измерять и проверять свои предположения.

Поскольку мы не пишем машинный код напрямую, трудно предсказать, как компилятор и JIT: er преобразуют вашу программу в машинный код. Вот почему нам нужно измерить время выполнения, чтобы убедиться, что мы ожидаем повышения производительности, и убедитесь, что фактическая программа не содержит никаких скрытых служебных данных.

Проверка - довольно простой процесс, который включает в себя обратное проектирование исполняемого файла, используя, например, инструменты, такие как ILSpy . JIT: er усложняет проверку, поскольку просмотр фактического машинного кода является сложным, но выполнимым. Однако, как правило, изучение `IL-code` дает большие выгоды.

Более сложная проблема - измерение; сложнее, потому что сложно настроить реалистичные ситуации, которые позволяют измерять улучшения в коде. Все еще измерение неоценимо.

Анализ простых функций F #

Рассмотрим некоторые простые функции `F#` которые накапливают все целые числа в `1..n` написанные различными способами. Поскольку диапазон является простой арифметической серией, результат может быть вычислен непосредственно, но для целей этого примера мы перебираем диапазон.

Сначала мы определим некоторые полезные функции для измерения времени, которое занимает функция:

```
// 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` запускает действие повторно, нам нужно запустить тесты на несколько сотен миллисекунд, чтобы уменьшить дисперсию.

Затем мы определяем несколько функций, которые накапливают все целые числа в `1..n` разными способами.

```
// 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
```

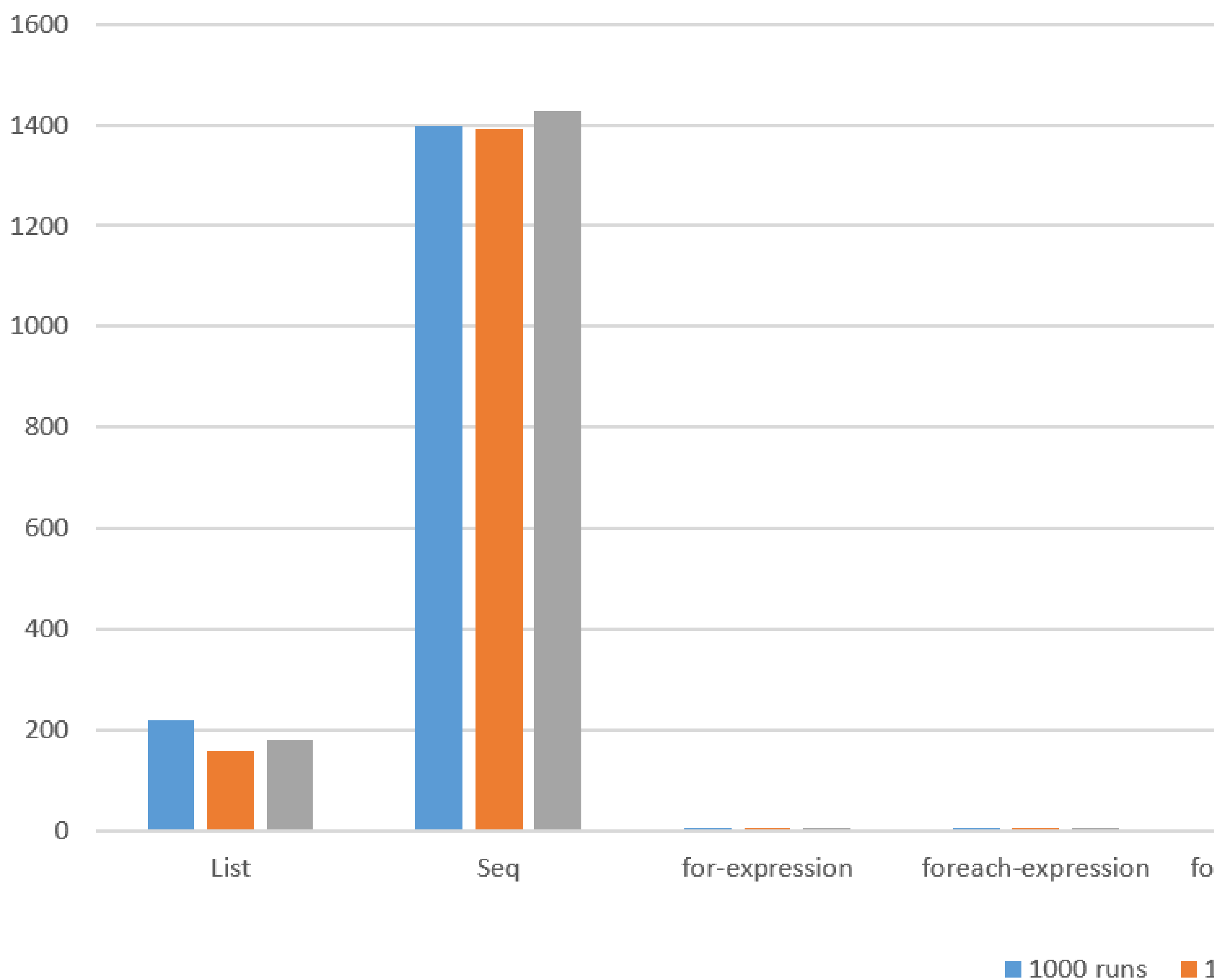
```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// 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
```

Мы предполагаем, что результат будет одинаковым (за исключением одной из функций, использующих приращение `2`), но есть разница в производительности. Для измерения этого определяется следующая функция:

```
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
```

Результат теста при работе на .NET 4.5.2 x64:



Мы видим драматическую разницу, и некоторые результаты неожиданно плохи.

Давайте посмотрим на плохие случаи:

Список

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

Здесь происходит полный список, содержащий все целые числа `1..n` создается и уменьшается с использованием суммы. Это должно быть более дорогостоящим, чем просто повторение и накопление в диапазоне, похоже, примерно на 42x медленнее, чем цикл for.

Кроме того, мы видим, что GC выполнял около 100x во время тестового прогона, потому что код выделял много объектов. Это также стоит CPU.

Seq

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

Версия `Seq` не выделяет полный `List` поэтому немного удивляет, что это ~ 270x медленнее, чем цикл for. Кроме того, мы видим, что GC выполнил 661x.

`Seq` неэффективен, когда количество работы на один элемент очень невелико (в этом случае агрегирование двух целых чисел).

Дело не в том, чтобы никогда не использовать `Seq` . Дело в том, чтобы измерить.

(**manofstick edit:** `Seq.init` является виновником этой серьезной проблемы с производительностью. Гораздо эффективнее использовать выражение `{ 0 .. n }` вместо `Seq.init (n+1) id` . Это станет намного более эффективным когда этот PR объединен и выпущен. Даже после релиза исходный `Seq.init ... |> Seq.sum` все равно будет медленным, но несколько конт-интуитивно, `Seq.init ... |> Seq.map id |> Seq.sum` будет довольно быстрым. Это должно было поддерживать обратную совместимость с реализацией `Seq.init` , которая не вычисляет `Current` изначально, а скорее обертывает их в `Lazy` объект, хотя это тоже должно немного улучшиться из-за это PR . Примечание для редактора: извините, это своего рода бессвязные заметки, но я не хочу, чтобы люди были отстранены от Seq, когда улучшение только за углом ... *Когда это время придет, было бы хорошо обновить диаграммы которые находятся на этой странице.*)

foreach-expression через список

```
// 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
```

Разница между этими двумя функциями очень тонкая, но разница в производительности не составляет примерно 76x. Зачем? Давайте перепроектировать плохой код:

```
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` реализован как эффективный в `while` цикла , но `for i in [1..n]` превращается в:

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

Это означает, что сначала мы создаем `Seq` над `1..n` и, наконец, `ToList` .

Дорого.

приращение foreach-expression 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
```

Еще раз разница между этими двумя функциями тонкая, но разница в производительности - жестокая: ~ 25x

Еще раз запустим `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;
}
```

A `Seq` создается над `1..2..n` а затем мы перебираем `Seq` с помощью перечислителя.

Мы ожидали, что `F#` создаст что-то вроде этого:

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

Однако компилятор `F#` поддерживает только эффективные для циклов по диапазонам int32, которые увеличиваются на единицу. Для всех остальных случаев он возвращается к `Operators.OperatorIntrinsics.RangeInt32` . Что объяснит следующий сюрпризный результат

foreach-expression более 64 бит

```
// 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
```

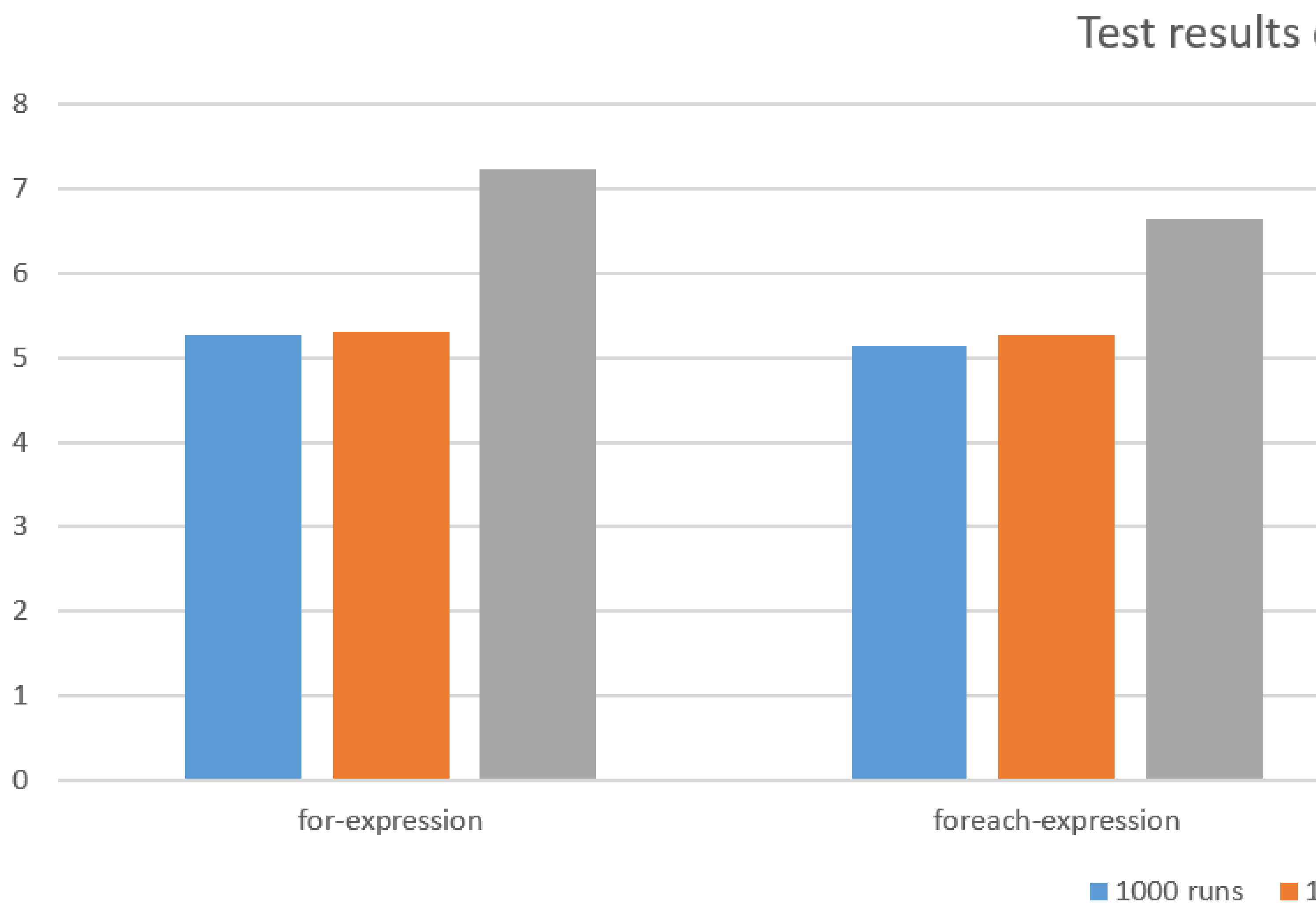
Это выполняет ~ 47 раз медленнее, чем цикл for, единственное отличие состоит в том, что мы перебираем более 64 битных целых чисел. `ILSpy` показывает нам, почему:

```
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#` поддерживает только эффективные для циклов для чисел `int32` он должен использовать резервные `Operators.OperatorIntrinsics.RangeInt64` .

Другие случаи примерно одинаковы:



Причина снижения производительности для более крупных тестовых прогонов заключается в том, что накладные расходы при вызове `action` растут, поскольку мы делаем все меньше и меньше работы в `action` .

Цикл в направлении `0` может иногда давать преимущества производительности, так как он может сохранить регистр CPU, но в этом случае CPU имеет резервные копии, поэтому он, похоже, не имеет значения.

Заключение

Измерение важно, потому что в противном случае мы могли бы подумать, что все эти альтернативы эквивалентны, но некоторые альтернативы медленнее, чем другие.

Шаг подтверждения включает обратное проектирование, исполняемый файл помогает нам объяснить, *почему* мы сделали или не достигли ожидаемой производительности. Кроме того, проверка может помочь нам прогнозировать производительность в случаях, когда слишком сложно выполнить правильное измерение.

Трудно прогнозировать производительность всегда. Измерьте, всегда проверяйте свои предположения о производительности.

Сравнение различных конвейеров данных F

В `F#` существует множество вариантов создания конвейеров данных, например: `List` , `Seq` и `Array` .

Какой конвейер данных предпочтительнее из использования памяти и производительности?

Чтобы ответить на это, мы сравним производительность и использование памяти с использованием разных конвейеров.

Конвейер данных

Чтобы измерить накладные расходы, мы будем использовать конвейер данных с низкой стоимостью процессора на обрабатываемые предметы:

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

Мы создадим эквивалентные конвейеры для всех альтернатив и сравним их.

Мы будем варьировать размер `n` но пусть общее число работ будет одинаковым.

Альтернативные источники данных

Мы сравним следующие альтернативы:

1. Императивный код
2. Массив (нелогичный)
3. Список (не ленивый)
4. LINQ (Lazy pull stream)
5. Seq (Lazy pull stream)
6. Нессос (ленивый трюк / толчок)

7. PullStream (упрощенный поток тяги)
8. PushStream (Упрощенный поток push)

Хотя это не конвейер данных, мы будем сравнивать с `Imperative` кодом, так как это наиболее точно соответствует тому, как процессор выполняет код. Это должен быть самый быстрый способ вычислить результат, позволяющий нам измерить накладные расходы на производительность для конвейеров данных.

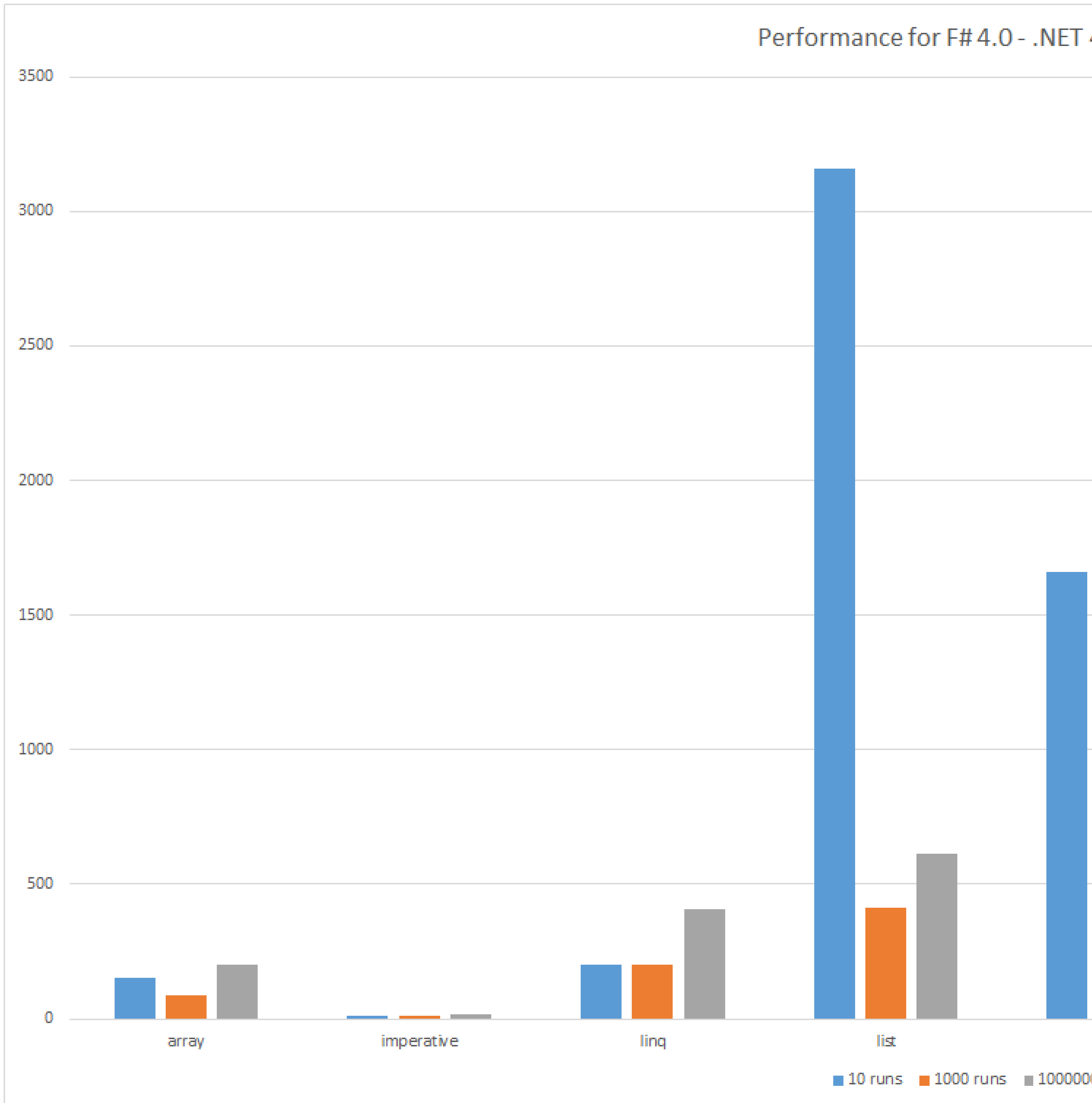
`Array` и `List` вычисляют полный `Array` / `List` на каждом шаге, поэтому мы ожидаем нехватки памяти.

`LINQ` и `Seq` основаны на `IEnumerable<'T>` который представляет собой ленивый поток pull (pull означает, что потребительский поток вытаскивает данные из потока производителя). Поэтому мы ожидаем, что производительность и использование памяти будут одинаковыми.

`Nessos` - это высокопроизводительная потоковая библиотека, которая поддерживает push и pull (например, Java `Stream`).

PullStream и PushStream являются упрощенными реализациями потоков `Pull` & `Push` .

Производительность Результаты работы: F # 4.0 - .NET 4.6.1 - x64



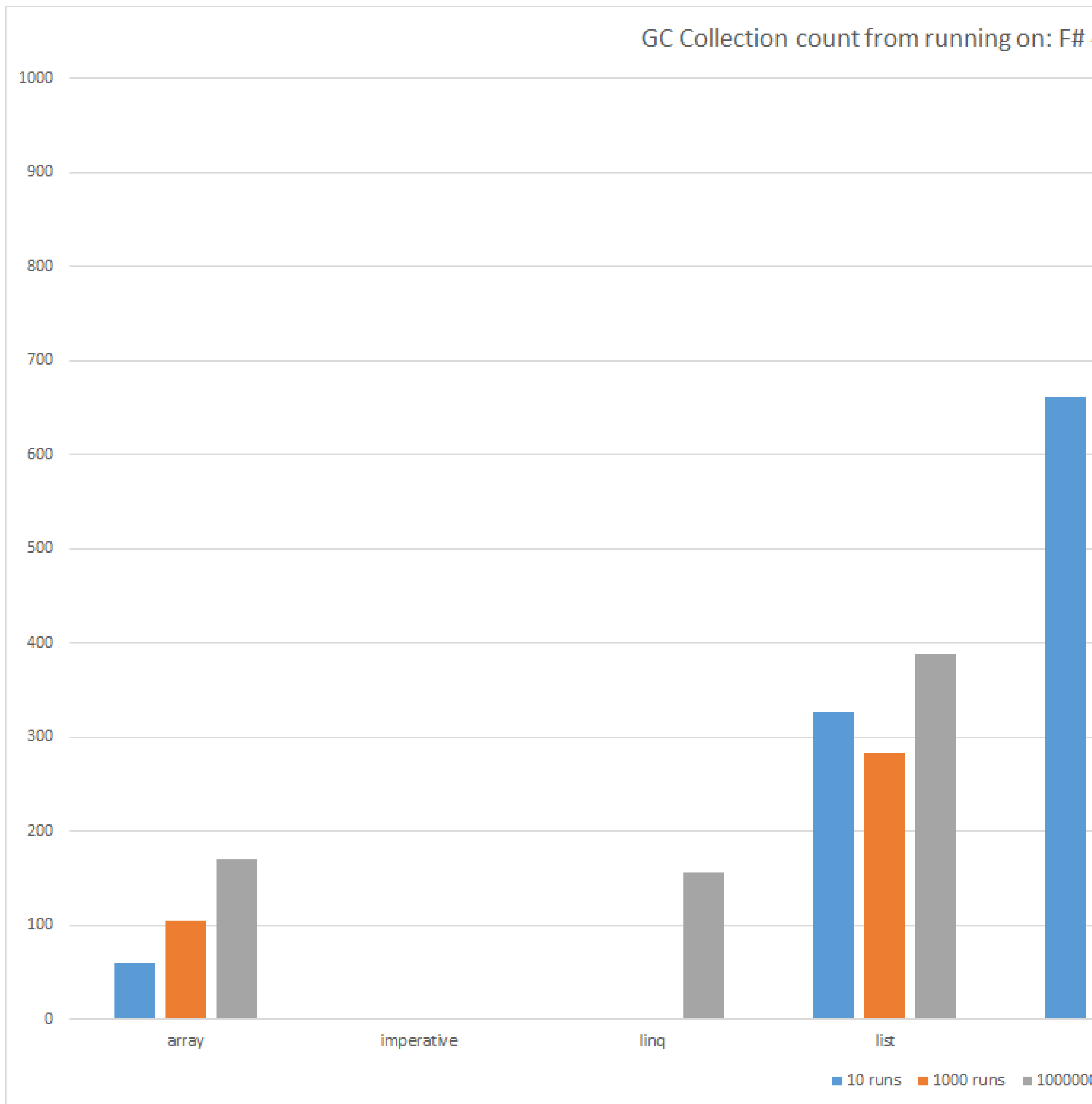
Бары показывают прошедшее время, лучше - ниже. Общий объем полезной работы для всех тестов одинаковый, поэтому результаты сопоставимы. Это также означает, что несколько запусков подразумевают большие наборы данных.

Как обычно, при измерении одни видят интересные результаты.

1. `List` показателей производительности сравнивается с другими альтернативами для больших наборов данных. Это может быть связано с `GC` или плохим местоположением кеша.
2. Производительность `Array` лучше, чем ожидалось.

3. `LINQ` работает лучше, чем `Seq` , это неожиданно, потому что оба они основаны на `IEnumerable<'T>` . Тем не менее, `Seq` внутренне основан на общей импиментации для всех алгоритмов, в то время как `LINQ` использует специализированные алгоритмы.
4. `Push` работает лучше, чем `Pull` . Это ожидается, так как в потоковом канале данных меньше проверок
5. Простейшие `Push` конвейеры данных сравнимы с `Nessos` . Однако `Nessos` поддерживает вытягивание и параллелизм.
6. Для небольших конвейеров данных производительность `Nessos` ухудшается, потому что накладные расходы на установку трубопроводов.
7. Как и ожидалось, `Imperative` код

GC Collection count from running on: F # 4.0 - .NET 4.6.1 - x64



Бары показывают общее количество подсчетов `GC` в ходе теста, более низкое. Это измерение количества объектов, созданных конвейером данных.

Как обычно, при измерении одни видят интересные результаты.

1. `List` как ожидается, создает больше объектов, чем `Array` потому что `List` - это по существу единственный связанный список узлов. Массив - это непрерывная область памяти.
2. Рассматривая базовые номера, `List` & `Array` создает 2 поколения коллекций. Такие коллекции дороги.

3. `Seq` запускает удивительное количество коллекций. Это удивительно даже хуже, чем `List` в этом отношении.
4. `LINQ` , `Nessos` , `Push` и `Pull` запускают коллекции для нескольких прогонов. Тем не менее, объекты выделяются, поэтому `GC` конечном итоге придется запускать.
5. Как и ожидалось, поскольку `Imperative` код не выделяет никаких объектов, не собирались коллекции `GC` .

Заключение

Все конвейеры данных выполняют одинаковое количество полезной работы во всех тестовых случаях, но мы видим значительные различия в производительности и использовании памяти между различными конвейерами.

Кроме того, мы замечаем, что накладные расходы на трубопроводы данных различаются в зависимости от размера обрабатываемых данных. Например, для небольших размеров `Array` работает довольно хорошо.

Следует иметь в виду, что объем работы, выполняемой на каждом этапе трубопровода, очень мал, чтобы измерить накладные расходы. В «реальных» ситуациях накладные расходы `Seq` могут не иметь значения, поскольку фактическая работа занимает больше времени.

Больше внимания вызывают различия в использовании памяти. `GC` не является свободным, и для долговременных приложений полезно поддерживать давление `GC` .

Для разработчиков `F#` связанных с производительностью и использованием памяти, рекомендуется проверить Nessos Streams .

Если вам нужна первоклассная производительность, стратегически размещенный `Imperative` код заслуживает рассмотрения.

Наконец, когда дело доходит до производительности, не делайте предположений. Измерять и проверять.

Полный исходный код:

```
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
```

Прочитайте F # Советы и подсказки производительности онлайн: https://riptutorial.com/ru/fsharp/topic/3562/f-sharp-советы-и-подсказки-производительности

глава 5: Активные шаблоны

Examples

Простые активные шаблоны

Активные модели представляют собой особый тип сопоставления с образцом , где вы можете указать , названные категории , что ваши данные могут попасть в, а затем использовать эти категории в `match` отчетности.

Чтобы определить активный шаблон, который классифицирует числа как положительные, отрицательные или нулевые:

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

Затем это можно использовать в выражении соответствия шаблону:

```
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
```

Активные шаблоны с параметрами

Активные шаблоны - это просто простые функции.

Подобно функциям вы можете определить дополнительные параметры:

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

Это можно использовать в шаблоне, соответствующем этому способу:

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

Активные шаблоны могут использоваться для проверки и преобразования аргументов функции

Интересное, но довольно неизвестное использование активных шаблонов в `F#` заключается в том, что они могут использоваться для проверки и преобразования аргументов функции.

Рассмотрим классический способ проверки правильности аргументов:

```
// 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 ()
```

Обычно мы добавляем код в метод, чтобы проверить правильность аргументов. Используя Active Patterns в `F#` мы можем обобщить это и объявить намерение в объявлении аргумента.

Следующий код эквивалентен приведенному выше коду:

```
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 ()
```

Для пользователя функций `f` и `g` нет разницы между двумя разными версиями.

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

Вызывает беспокойство, если Active Patterns увеличивает накладные расходы. Давайте используем `ILSpy` для декомпиляции `f` и `g` чтобы убедиться, что это так.

```
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();
}
```

Благодаря `inline` Active Patterns добавляет лишних накладных расходов по сравнению с классическим способом проверки правильности аргументов.

Активные шаблоны как обертки .NET API

Активные шаблоны могут использоваться для того, чтобы сделать вызов некоторого .NET API более естественным, особенно те, которые используют выходной параметр, чтобы возвращать больше, чем просто возвращаемое значение функции.

Например, вы обычно вызываете метод `System.Int32.TryParse` следующим образом:

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

Вы можете немного улучшить это, используя сопоставление шаблонов:

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

Однако мы также можем определить следующий активный шаблон, который обертывает функцию `System.Int32.TryParse` :

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

Теперь мы можем сделать следующее:

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

Другим хорошим кандидатом для включения в активные шаблоны являются API регулярных выражений:

```
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"
```

Полные и частичные активные шаблоны

Существует два типа активных шаблонов, которые несколько отличаются в использовании - Полный и Частичный.

Полные активные шаблоны можно использовать, когда вы можете перечислить все результаты, например «число нечетное или даже?».

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

Обратите внимание, что определение активного шаблона содержит список возможных случаев и ничего другого, и тело возвращает один из перечисленных случаев. Когда вы используете его в выражении соответствия, это полное совпадение:

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

Это удобно, если вы хотите разбить входное пространство на известные категории, которые полностью его покрывают.

С другой стороны, частичные активные шаблоны позволяют явно игнорировать некоторые возможные результаты, возвращая `option` . В их определении используется частный случай _ для непревзойденного случая.

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

Таким образом, мы можем сопоставлять, даже если некоторые случаи не могут быть обработаны нашей функцией синтаксического анализа.

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

Частичные активные шаблоны могут использоваться как форма тестирования, независимо от того, попадает ли вход в определенную категорию во входное пространство, игнорируя другие параметры.

Прочитайте Активные шаблоны онлайн: https://riptutorial.com/ru/fsharp/topic/962/активные-шаблоны

глава 6: Введение в WPF в F

Вступление

В этом разделе показано, как использовать **функциональное программирование** в **приложении WPF** . Первый пример - это сообщение от Māris Krivtežs (см. Раздел « *Примечания* » внизу). Причина пересмотра этого проекта двоякая:

1 \ Конструкция поддерживает разделение проблем, в то время как модель поддерживается чистой, а изменения распространяются функционально.

2 \ Сходство приведет к легкому переходу на реализацию Gjallarhorn.

замечания

Демо-проекты библиотеки @GitHub

- FSharp.ViewModule (под FsXaml)
- Gjallarhorn (ref Образцы)

Марис Кривтеж написал две большие посты на эту тему:

- Приложение F # XAML - MVVM и MVC, где выделены плюсы и минусы обоих подходов.

  > Я чувствую, что ни один из этих стилей приложений XAML не выигрывает от функционального программирования. Я предполагаю, что идеальное приложение будет состоять из представления, которое создает события и события, имеющие текущее состояние представления. Вся прикладная логика должна обрабатываться путем фильтрации и манипулирования событиями и моделью представления, а на выходе она должна создать новую модель представления, которая привязана к представлению.

- F # XAML - MVVM с ведомым событием, рассмотренным выше.

Examples

FSharp.ViewModule

Наше демо-приложение состоит из табло. Модель оценки - неизменная запись. События табло содержатся в типе Союза.

```
namespace Score.Model

type Score = { ScoreA: int ; ScoreB: int }
```

```
type ScoringEvent = IncA | DecA | IncB | DecB | NewGame
```

Изменения распространяются путем прослушивания событий и соответственно изменения модели представления. Вместо добавления членов в тип модели, как и в ООП, мы объявляем отдельный модуль для размещения разрешенных операций.

```
[<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
```

Наша модель представления `EventViewModelBase<'a>` из `EventViewModelBase<'a>` , которая имеет свойство `EventStream` типа `IObservable<'a>` . В этом случае события, которые мы хотим подписаться, имеют тип `ScoringEvent` .

Контроллер управляет событиями в функциональном режиме. Его подпись `Score -> ScoringEvent -> Score` показывает нам, что всякий раз, когда происходит событие, текущее значение модели преобразуется в новое значение. Это позволяет нашей модели оставаться чистым, хотя наша модель взглядов не является.

`eventHandler` отвечает за изменение состояния представления. Наследуя от `EventViewModelBase<'a>` мы можем использовать `EventValueCommand` и `EventValueCommandChecked` для подключения событий к командам.

```
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
```

Код файла (* .xaml.fs) - это место, где все сведено вместе, т.е. функция обновления (`controller`) вводится в `MainViewModel` .

```
namespace Score.Views

open FsXaml

type MainView = XAML<"MainWindow.xaml">

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

Тип `CompositionRoot` является оберткой, на которую ссылается файл XAML.

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

Я больше не буду погружаться в файл XAML, поскольку это базовый материал WPF, весь проект можно найти на GitHub .

Gjallarhorn

Основные типы библиотеки Gjallarhorn реализуют `IObservable<'a>` , что сделает реализацию знакомой (помните свойство `EventStream` из примера FSharp.ViewModule). Единственное реальное изменение нашей модели - это порядок аргументов функции обновления. Кроме того, теперь мы используем термин « *Сообщение* вместо *события»* .

```
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
```

Чтобы создать пользовательский интерфейс с Gjallarhorn, вместо того, чтобы создавать

классы для поддержки привязки данных, мы создаем простые функции, называемые `Component` . В их конструкторе первый `source` аргумента имеет тип `BindingSource` (определенный в Gjallarhorn.Bindable) и используется для сопоставления модели с представлением и событий из представления обратно в сообщения.

```
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
        ]
```

Текущая реализация отличается от версии FSharp.ViewModule тем, что две команды не имеют CanExecute должным образом реализованы. Также перечисление сантехники приложения.

```
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
```

Осталось настроить развязанный вид, объединяя тип `MainWindow` и логическое приложение.

```
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
```

Это подводит итог основным концепциям, дополнительной информации и более сложному примеру, пожалуйста, обратитесь к сообщению Рида Копси . Проект « *Рождественские деревья* » подчеркивает пару преимуществ такого подхода:

- Эффективно избавляя нас от необходимости (вручную) копировать модель в пользовательскую коллекцию моделей взглядов, управлять ими и вручную конструировать модель из результатов.
- Обновления в коллекциях выполняются прозрачным образом, сохраняя чистую модель.
- Логика и представление организованы двумя разными проектами, что подчеркивает

разделение проблем.

Прочитайте Введение в WPF в F # онлайн: https://riptutorial.com/ru/fsharp/topic/8758/введение-в-wpf-в-f-sharp

глава 7: Дженерики

Examples

Сторнирование списка любого типа

Чтобы отменить список, не важно, какие типы являются элементами списка, только то, в каком порядке они находятся. Это идеальный кандидат на универсальную функцию, поэтому можно использовать ту же самую обратную функцию независимо от того, какой список передан.

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

В коде не делается никаких предположений о типах элементов. Компилятор (или F # interactive) скажет вам, что сигнатура типа этой функции представляет собой `'T list -> 'T list` `'T` сообщает вам, что это общий тип без ограничений. Вы также можете увидеть `'a` вместо `'T` - письмо несущественно, потому что это всего лишь *общий* заполнитель. Мы можем передать `int list` или `int list string list` , и оба они будут успешно работать, возвращая `int list` или `string list` соответственно.

Например, в 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"]
```

Отображение списка в другом виде

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

Подписями этой функции является `('a -> 'b) -> 'a list -> 'b list` , который является наиболее общим. Это не мешает `'a` быть тем же типом, что и `'b` , но также позволяет им быть другим. Здесь вы можете видеть, что `'a` тип, являющийся параметром функции `f` должен соответствовать типу параметра `list` . Эта функция по-прежнему носит общий характер, но есть некоторые небольшие ограничения на входы - если типы не совпадают,

будет ошибка компиляции.

Примеры:

```
> 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'
```

Прочитайте Дженерики онлайн: https://riptutorial.com/ru/fsharp/topic/7731/дженерики

глава 8: Дискриминационные союзы

Examples

Именование элементов кортежей в рамках дискриминационных союзов

При определении дискриминационных союзов вы можете назвать элементы типов кортежей и использовать эти имена во время сопоставления шаблонов.

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

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

Кроме того, именование элементов дискриминационных объединений улучшает читаемость кода и совместимость с именами, указанными в C #, будут использоваться для имен свойств и параметров конструкторов. По умолчанию сгенерированные имена в коде взаимодействия являются «Item», «Item1», «Item2» ...

Использование основного дискриминационного союза

Дискриминационные союзы в F # предлагают способ определения типов, которые могут содержать любое количество разных типов данных. Их функциональность подобна объединениям C ++ или вариантам VB, но с дополнительным преимуществом безопасного типа.

```
// 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
```

Объединения в стиле enum

Информация о типе не требуется включать в случаи дискриминационного союза. Опуская информацию о типе, вы можете создать объединение, которое просто представляет собой набор вариантов, аналогично перечислению.

```
// 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
```

Преобразование в строки и из строк с отражением

Иногда необходимо преобразовать дискриминированный союз в строку и из нее:

```
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
```

Единый случай дискриминации

Единственный случай, когда дискриминационный союз подобен любому другому дискриминационному союзу, за исключением того, что он имеет только один случай.

```
// 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
```

Он полезен для обеспечения безопасности типов и обычно используется в F #, в отличие от C # и Java, где создание новых типов требует дополнительных накладных расходов.

Следующие два альтернативных типа определения приводят к объявлению одного и того же объявленного единственного случая:

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

Использование дискретизированных отдельных случаев в качестве

записей

Иногда бывает полезно создавать типы объединения только с одним случаем для реализации типа записей:

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)

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

Они становятся очень полезными, потому что их можно разложить с помощью сопоставления с образцом так же, как аргументы кортежа:

```
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

С атрибутом `RequireQualifiedAccess` , случаи объединения должны упоминаться как `MyUnion.MyCase` а не только `MyCase` . Это предотвращает конфликты имен во вмещающем пространстве имен или модуле:

```
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
```

Если, например, `System` была открыта, `Single` относится к `System.Single` . Нет столкновения с профсоюзным случаем. `Requirements.Single` .

Рекурсивные дискриминационные союзы

Рекурсивный тип

Дискриминационные союзы могут быть рекурсивными, то есть они могут ссылаться на себя в своем определении. Главным примером здесь является дерево:

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

В качестве примера давайте определим следующее дерево:

```
    1
  2   5
3   4
```

Мы можем определить это дерево, используя наш рекурсивный дискриминированный союз следующим образом:

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

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

Итерация по дереву - это просто вопрос соответствия шаблонов:

```
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]
```

Взаимно зависимые рекурсивные типы

Одним из способов достижения рекурсии является наличие вложенных взаимозависимых типов.

```
// 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
```

Определение типа записи непосредственно внутри дискриминационного объединения устарело:

```
// BAD
type Expression =
```

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

Вы можете использовать `and` ключевое слово для цепочки взаимозависимых определений:

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

Прочитайте Дискриминационные союзы онлайн: https://riptutorial.com/ru/fsharp/topic/1025/дискриминационные-союзы

глава 9: документация

Examples

Добавить функции-члены в записи

```
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
```

Основное использование

```
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
```

Прочитайте документация онлайн: https://riptutorial.com/ru/fsharp/topic/1136/документация

глава 10: Единицы измерения

замечания

Единицы во время выполнения

Единицы измерения используются только для статической проверки компилятором и не доступны во время выполнения. Они не могут использоваться в отражении или в методах, подобных `ToString` .

Например, C # дает `double` без единиц для поля типа `float<m>` определенного и отображаемого из библиотеки F #.

Examples

Обеспечение согласованных единиц в расчетах

Единицы измерения - это дополнительные аннотации типа, которые могут быть добавлены к поплавкам или целым числам. Они могут использоваться для проверки во время компиляции, когда расчеты используют единицы последовательно.

Чтобы определить аннотации:

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

После определения аннотации могут использоваться для проверки того, что выражение приводит к ожидаемому типу.

```
// 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
```

Конверсии между единицами

```
[<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>
```

Обратите внимание, что компилятор F # не знает, что `1<m>` равно `100<cm>` . Что касается этого, единицы являются отдельными типами. Вы можете написать аналогичные функции для преобразования от метров до килограммов, и компилятору все равно.

```
[<Measure>] type kg

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

Невозможно определить единицы измерения как кратность других единиц, таких как

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

Однако для определения единиц «за что-то», например, Герца, измеряя частоту, просто «в секунду», довольно просто.

```
// 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
```

Использование LanguagePrimitives для сохранения или установки единиц

Когда функция не сохраняет единицы автоматически из-за операций более низкого уровня, модуль `LanguagePrimitives` может использоваться для установки единиц на примитивах, которые их поддерживают:

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

Чтобы присвоить единицы измерения значению с плавающей запятой с двойной точностью, просто умножьте их на единицы с правильными единицами:

```
[<Measure>]
type USD

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

Чтобы присвоить единицы измерения единичному значению, которое не является System.Double, например, поступая из библиотеки, написанной на другом языке, используйте преобразование:

```
open LanguagePrimitives

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

Ниже приведены типы функций, сообщенные F # interactive:

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

Параметры типа единицы измерения

Атрибут `[<Measure>]` может использоваться для параметров типа для объявления типов, которые являются общими для единиц измерения:

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

Использование теста:

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

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

Используйте стандартные типы устройств для обеспечения совместимости

Например, типы для единиц СИ были стандартизованы в основной библиотеке F # в

`Microsoft.FSharp.Data.UnitSystems.SI` . Чтобы использовать их, откройте соответствующее пространство `UnitNames` , `UnitNames` или `UnitSymbols` . Или, если требуется только несколько единиц СИ, их можно импортировать с помощью псевдонимов типов:

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

Некоторые пользователи имеют тенденцию делать следующее, что **не должно выполняться** всякий раз, когда определение уже доступно:

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

Разница становится очевидной при взаимодействии с другим кодом, который относится к стандартным типам SI. Код, который относится к стандартным единицам, совместим, а код, который определяет его собственный тип, несовместим с любым кодом, не использующим его конкретное определение.

Поэтому всегда используйте стандартные типы для единиц СИ. Не имеет значения, ссылаетесь ли вы на `UnitNames` или `UnitSymbols` , поскольку эквивалентные имена в этих двух относятся к одному типу:

```
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>
```

Прочитайте Единицы измерения онлайн: https://riptutorial.com/ru/fsharp/topic/1055/единицы-измерения

глава 11: Использование F #, WPF, FsXaml, меню и диалогового окна

Вступление

Целью здесь является создание простого приложения в F # с использованием Windows Presentation Foundation (WPF) с традиционными меню и диалоговыми окнами. Это связано с моим разочарованием в попытке пробраться через сотни разделов документации, статей и сообщений, посвященных F # и WPF. Чтобы что-либо делать с WPF, вам, похоже, все нужно знать об этом. Моя цель заключается в том, чтобы предоставить возможный путь, простой настольный проект, который может служить шаблоном для ваших приложений.

Examples

Настроить проект

Предположим, вы делаете это в Visual Studio 2015 (сообщество VS 2015, в моем случае). Создайте пустой проект консоли в VS. В проекте | Свойства изменяют тип вывода на приложение Windows.

Затем используйте NuGet для добавления FsXaml.Wpf в проект; этот пакет был создан оценочным Ридом Копси, младший, и он значительно упрощает использование WPF из F #. При установке он добавит ряд других сборок WPF, поэтому вам не придется. В FsXaml есть другие подобные пакеты, но одна из моих целей заключалась в том, чтобы максимально сократить количество инструментов, чтобы сделать общий проект максимально простым и доступным.

Кроме того, добавьте UIAutomationTypes в качестве ссылки; он входит в состав .NET.

Добавьте «Бизнес-логику»

Предположительно, ваша программа что-то сделает. Добавьте свой рабочий код в проект вместо Program.fs. В этом случае наша задача - нарисовать кривые спирографа на холсте окна. Это выполняется с использованием Spirograph.fs, ниже.

```
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 - это первый файл F # в порядке компиляции, поэтому он содержит определения типов, которые нам понадобятся. Его задача - нарисовать спирограф в главном окне Canvas на основе параметров, введенных в диалоговом окне. Поскольку есть много ссылок на то, как нарисовать спирограф, мы не будем здесь заниматься этим.

Создайте главное окно в XAML

Вам необходимо создать файл XAML, который определяет главное окно, содержащее наше меню и пространство рисования. Вот код XAML в 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>
```

Комментарии, как правило, не включены в файлы XAML, что, я думаю, является ошибкой. Я добавил некоторые комментарии ко всем файлам XAML в этом проекте. Я не утверждаю, что они лучшие комментарии, когда-либо написанные, но они хотя бы показывают, как комментарий должен быть отформатирован. Обратите внимание, что вложенные комментарии не допускаются в XAML.

Создайте диалоговое окно в XAML и F

Файл XAML для параметров спирографа приведен ниже. Он включает в себя три текстовых поля для параметров спирографа и группу из трех переключателей для цвета. Когда мы даем радиокнопкам одно и то же имя группы - как здесь, WPF обрабатывает включение / выключение при выборе одного из них.

```
<!-- 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>
```

Теперь мы добавим код для Dialog.Box. По соглашению, код, используемый для обработки интерфейса диалогового окна с остальной частью программы, называется XXX.xaml.fs, где связанный файл XAML называется 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
    ()
```

Большая часть кода здесь посвящена обеспечению того, чтобы параметры спирографа в Spirograph.fs соответствовали параметрам, показанным в этом диалоговом окне. Обратите внимание, что проверка ошибок отсутствует. Если вы введете плавающую точку для целых чисел, ожидаемых в двух верхних полях параметров, программа выйдет из строя. Поэтому, пожалуйста, добавьте проверку ошибок в свои собственные усилия.

Также обратите внимание, что поля ввода параметров отключены, и в переключателях выбран случайный цвет. Здесь просто показать, как это можно сделать.

Чтобы переместить данные назад и вперед между диалоговым окном и программой, я использую System.Windows.Element.FindName (), чтобы найти соответствующий элемент управления, применить его к элементу управления, которым он должен быть, а затем получить соответствующие настройки из Контроль. Большинство других примеров программ используют привязку данных. Я не по двум причинам: во-первых, я не мог понять, как заставить его работать, а во-вторых, когда это не сработало, я не получил никаких сообщений об ошибках. Возможно, кто-то, кто посещает это в StackOverflow, может сказать мне, как использовать привязку данных, не включая совершенно новый набор пакетов NuGet.

Добавьте код для 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
```

Здесь не так много: при необходимости мы открываем диалоговое окно «Параметры», и у нас есть возможность перерисовать спирограф с любыми текущими параметрами.

Добавьте App.xaml и App.xaml.fs, чтобы связать все вместе

```
<!-- 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>
```

Вот код:

```
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 - это все здесь, в основном, чтобы показать, где могут быть объявлены ресурсы приложений, такие как значки, графика или внешние файлы. Сопутствующий App.xaml.fs объединяет модель и MainWindow, размеры MainWindow до двух третей доступного размера экрана и запускает его.

Когда вы создадите это, не забудьте убедиться, что для свойства Build для каждого xaml-файла установлено значение Resource. Затем вы можете либо запустить отладчик, либо выполнить компиляцию в exe-файл. Обратите внимание, что вы не можете запустить это с помощью интерпретатора F #: пакет FsXaml и интерпретатор несовместимы.

Там у вас это есть. Надеюсь, вы сможете использовать это как отправную точку для своих приложений, и при этом вы можете расширить свои знания за пределы того, что показано здесь. Будут оценены любые комментарии и предложения.

Прочитайте Использование F #, WPF, FsXaml, меню и диалогового окна онлайн: https://riptutorial.com/ru/fsharp/topic/9145/использование-f-sharp--wpf--fsxaml--меню-и-диалогового-окна

глава 12: Классы

Examples

Объявление класса

```
// 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
```

Создание экземпляра

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

Прочитайте Классы онлайн: https://riptutorial.com/ru/fsharp/topic/3003/классы

глава 13: Ленивая оценка

Examples

Ленивая оценка

Большинство языков программирования, включая F #, оценивают вычисления немедленно в соответствии с моделью под названием Strict Evaluation. Однако в Lazy Evaluation вычисления не оцениваются до тех пор, пока они не понадобятся. F # позволяет использовать ленивую оценку как по `lazy` ключевому слову, так и по `sequences` .

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

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

Кроме того, при использовании Lazy Evaluation результаты вычисления кэшируются, поэтому, если мы вынуждаем результат после нашего первого экземпляра принудительного его применения, само выражение не будет оцениваться снова

```
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()
```

Введение в ленивую оценку в F

F #, как и большинство языков программирования, по умолчанию использует Strict Evaluation. При строгой оценке вычисления выполняются немедленно. Напротив, Lazy Evaluation, откладывает выполнение вычислений до тех пор, пока их результаты не будут необходимы. Более того, результаты вычисления в рамках «ленивой оценки» кэшируются, тем самым устраняя необходимость переоценки выражения.

Мы можем использовать оценку Lazy в F # через как `lazy` ключевое слово, так и `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()
```

Прочитайте Ленивая оценка онлайн: https://riptutorial.com/ru/fsharp/topic/3682/ленивая-оценка

глава 14: мемоизации

Examples

Простое напоминание

Воспоминание состоит из результатов функции кеширования, чтобы избежать вычисления одного и того же результата несколько раз. Это полезно при работе с функциями, выполняющими дорогостоящие вычисления.

В качестве примера мы можем использовать простую факториальную функцию:

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

    innerLoop index 1
```

Вызов этой функции несколько раз с тем же параметром приводит к тому же вычислению снова и снова.

Воспоминание поможет нам кэшировать факторный результат и вернуть его, если один и тот же параметр будет передан снова.

Вот простая реализация memoization:

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

Функция `memoization` просто принимает функцию как параметр и возвращает функцию с той же сигнатурой. Это можно увидеть в сигнатуре метода `f:('a -> 'b) -> ('a -> 'b)` . Таким

образом, вы можете использовать memoization так же, как если бы вы вызывали факториальный метод.

`printfn` должны показывать, что происходит, когда мы вызываем функцию несколько раз; они могут быть удалены безопасно.

Использование memoization очень просто:

```
// 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
```

Воспоминание в рекурсивной функции

Используя предыдущий пример вычисления факториала целого числа, поместите в хеш-таблицу все значения факториала, вычисленные внутри рекурсии, которые не отображаются в таблице.

Как и в статье о memoization , мы объявляем функцию `f` которая принимает `fact` параметра функции и целочисленный параметр. Эта функция `f` включает в себя инструкции для вычисления факториала `n` из `fact(n-1)` .

Это позволяет обрабатывать рекурсию функцией, возвращаемой `memorec` а не самим `fact` , и, возможно, останавливать вычисление, если значение `fact(n-1)` уже рассчитано и находится в хеш-таблице.

```
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
```

Результат:

```
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
```

Прочитайте мемоизации онлайн: https://riptutorial.com/ru/fsharp/topic/2698/мемоизации

глава 15: Монады

Examples

Понимание Монад происходит из практики

Эта тема предназначена для разработчиков с промежуточными и продвинутыми F #

«Что такое Монады?» это общий вопрос. Это легко ответить, но, как и в путеводителях автостопом по галактике, мы понимаем, что не понимаем ответа, потому что мы не знали, о чем мы просим.

Многие считают, что путь к пониманию Монад - это практиковать их. Как программисты, мы, как правило, не заботимся о математической основе того, что такое Принцип замещения Лискова, подтипы или подклассы. Используя эти идеи, мы приобрели интуицию для того, что они представляют. То же самое верно для Монад.

Чтобы помочь вам начать работу с Monads, этот пример демонстрирует, как создать библиотеку Monadic Parser Combinator . Это может помочь вам начать работу, но понимание придет от написания вашей собственной библиотеки Monadic.

Достаточно проза, время для кода

Тип Parser:

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

Используя это определение Парсера, определим некоторые фундаментальные функции парсера

```
// 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` - функция, которая задает `sat` функцию, создает синтаксический анализатор, который преуспевает, если мы не передали `EOS` а символ в текущей позиции передает функцию `sat` . Используя функцию `satisfy` мы создаем ряд полезных парсеров символов.

Выполнение этого в 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)
```

У нас есть фундаментальные синтаксические анализаторы. Мы объединим их в более мощные синтаксические анализаторы, используя функции парсера-комбинатора

```
// '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)
```

Имена и подписи не выбраны произвольно, но мы не будем разбираться в этом, вместо этого посмотрим, как мы используем `bind` для объединения парсера в более сложные:

```
> 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)
```

Это показывает нам, что `bind` позволяет объединить два парсера в более сложный парсер.

В результате `bind` является парсером, который, в свою очередь, может быть объединен снова.

```
> 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` будет основным способом объединения парсеров, хотя мы будем определять вспомогательные функции для упрощения синтаксиса.

Одной из вещей, которые могут упростить синтаксис, являются выражения вычислений . Их легко определить:

```
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)
```

Это эквивалентно:

```
> 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)
```

Другим фундаментальным комбинатором парсеров, который мы будем использовать alot, является `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
```

Это позволяет нам определить `letterOrDigit` следующим образом:

```
> 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)
```

Замечание об операциях Infix

Общей проблемой FP является использование необычных инфиксных операторов, таких как `>>=` , `>=>` , `<-` и т. Д. Однако большинство из них не обеспокоено использованием `+` , `-` , `*` , `/` и `%` , это хорошо известные операторы, используемые для создания значений. Однако большая часть в FP заключается в составлении не только значений, но и функций. Для промежуточного разработчика FP операторы infix `>>=` , `>=>` , `<-` хорошо известны и должны иметь определенные подписи, а также семантику.

Для тех функций, которые мы определили до сих пор, мы определяли бы следующие инфиксные операторы, используемые для объединения парсеров:

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

Итак `>>=` означает `bind` и `<|>` означает `orElse` .

Это позволяет нам комбинировать парсеры более кратко:

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

Чтобы определить некоторые продвинутые компараторы парсеров, которые позволят нам анализировать более сложные выражения, мы определяем еще несколько простых комбинаторов парсеров:

```
// '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
  }
```

Мы готовы определить `many` и более `sepBy` которые являются более продвинутыми, поскольку они применяют входные парсеры, пока они не `sepBy` . Затем `many` и `sepBy` возвращают агрегированный результат:

```
// '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
```

Создание простого парсера выражений

С помощью созданных нами инструментов теперь мы можем определить парсер для простых выражений типа `1+2*3`

Мы начинаем снизу, определяя синтаксический анализатор для целых чисел `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)
  }
```

Мы пытаемся разобрать столько цифр, сколько можем, результат - `char list` . Если список пуст, мы `fail` , иначе мы сбрасываем символы в целое число.

Тестирование `pint` в FSI:

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

Кроме того, нам нужно проанализировать различные типы операторов, используемые для комбинирования целочисленных значений:

```
// 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)
```

Связывание всего этого:

```
// '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
  }
```

Выполнение всего этого в FSI:

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

Заключение

Определив `Parser<'T>` , `return_` , `bind` и убедившись, что они подчиняются монадическим законам, мы создали простую, но мощную структуру Monadic Parser Combinator.

Монады и парсеры идут вместе, потому что парсеры исполняются в состоянии парсера. Монады позволяют комбинировать парсеры, скрывая состояние парсера, тем самым уменьшая беспорядок и улучшая композиционную способность.

Созданная нами структура медленна и не вызывает сообщений об ошибках, чтобы сохранить код коротким. FParsec обеспечивает как приемлемую производительность, так и отличные сообщения об ошибках.

Однако один пример не может дать понимания Монадам. Нужно практиковать Монады.

Вот несколько примеров на Monads, которые вы можете попытаться реализовать, чтобы достичь достигнутого понимания:

1. State Monad - разрешает скрытое состояние окружающей среды неявно
2. Tracer Monad - позволяет отслеживать состояние неявно. Вариант государственной монады

3. Черепаха Монада - Монада для создания программ черепахи (Логосов). Вариант государственной монады
4. Продолжение Монада - сопрограмма Монады. Примером этого является `async` в F #

Лучше всего научиться придумывать приложение для Monads в домене, с которым вам удобно. Для меня это были парсеры.

Полный исходный код:

```
// 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
  }
```

Вычисления Выражения предоставляют альтернативный синтаксис для цепи Monads

К Monads относятся выражения вычисления `F#` (`CE`). Программист обычно реализует `CE` чтобы обеспечить альтернативный подход к цепочке Monads, т. Е. Вместо этого:

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

Вы можете написать это:

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

Оба стиля эквивалентны, и предпочтение разработчика зависит от выбора.

Чтобы продемонстрировать, как реализовать `CE` вам нравится, чтобы все следы включали идентификатор корреляции. Этот идентификатор корреляции поможет сопоставить трассировки, относящиеся к одному и тому же вызову. Это очень полезно, когда есть файлы журналов, содержащие следы от одновременных вызовов.

Проблема заключается в том, что в качестве аргумента ко всем функциям сложно включить идентификатор корреляции. Поскольку Monads позволяет переносить неявное состояние, мы определим Log Monad, чтобы скрыть контекст журнала (т.е. идентификатор корреляции).

Начнем с определения контекста журнала и типа функции, которая отслеживает контекст

журнала:

```
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 ())
```

Мы также определяем две функции трассировки, которые будут регистрироваться с идентификатором корреляции из контекста журнала:

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

`trace` - это `Function<unit>` которая означает, что при вызове будет передан контекст журнала. Из контекста журнала мы получаем идентификатор корреляции и отслеживаем его вместе с `v`

Кроме того, мы определяем `bind` и `return_` и поскольку они следуют Законам Монады, это формирует нашу 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
```

Наконец, мы определяем `LogBuilder` , который позволит нам использовать синтаксис `CE` для `LogBuilder 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 ()
```

Теперь мы можем определить наши функции, которые должны иметь неявный лог-контекст:

```
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
  }
```

Мы выполняем g с:

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

Какие принты:

```
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
```

Обратите внимание, что CorrelationId неявно переносится из `run` to `g` в `f` что позволяет нам сопоставлять записи журнала во время устранения проблем.

`CE` имеет намного больше возможностей, но это должно помочь вам приступить к разработке собственного `CE` : s.

Полный код:

```
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
```

Прочитайте Монады онлайн: https://riptutorial.com/ru/fsharp/topic/3320/монады

глава 16: операторы

Examples

Как составлять значения и функции с помощью общих операторов

В объектно-ориентированном программировании общей задачей является создание объектов (значений). В функциональном программировании также является общей задачей создавать значения, а также функции.

Мы привыкли составлять ценности из нашего опыта других языков программирования, используя такие операторы, как `+` , `-` , `*` , `/` и т. Д.

Составляющая стоимости

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

Поскольку функциональное программирование составляет функции, а также значения, неудивительно, что существуют общие операторы для составления функций, такие как `>>` , `<<` , `|>` и `<|` ,

Состав функции

```
// 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')
```

В `F#` прямой трубопровод является предпочтительным по сравнению с обратным трубопроводом, потому что:

1. Вывод типа (обычно) перемещается слева направо, так что естественно, что значения и функции также текут слева направо
2. Потому что `<|` и `<<` должны иметь правую ассоциативность, но в `F#` они остаются ассоциативными, что заставляет нас вставлять ()
3. Смешивание прямого и обратного трубопроводов обычно не работает, поскольку они имеют одинаковый приоритет.

Композиция Монады

Поскольку Monads (например, `Option<'T>` или `List<'T>`) обычно используются в функциональном программировании, также существуют обычные, но менее известные операторы для компоновки функций, работающих с Monads, таких как `>>=` , `>=>` , `<|>` и `<*>` .

```
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
```

Заключение

Для новых функциональных программистов состав функций с использованием операторов может казаться непрозрачным и неясным, но это потому, что смысл этих операторов не так широко известен как операторы, работающие над значениями. Однако при некоторой тренировке с использованием `|>` , `>>` , `>>=` и `>=>` становится таким же естественным, как использование `+` , `-` , `*` и `/` .

Latebinding в F # с использованием? оператор

На статически типизированном языке, таком как `F#` мы работаем с типами, хорошо известными во время компиляции. Мы потребляем внешние источники данных безопасным типом, используя поставщиков типов.

Тем не менее, иногда необходимо использовать позднюю привязку (например, `dynamic` на `C#`). Например, при работе с документами `JSON` , у которых нет четко определенной схемы.

Чтобы упростить работу с поздним связыванием, `F#` обеспечивает поддержку операторов динамического поиска `?` и `?<-` .

Пример:

```
// (?) 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
```

Оказывается, поддержка `F#` для позднего связывания проста, но гибкая.

Прочитайте операторы онлайн: https://riptutorial.com/ru/fsharp/topic/4641/операторы

глава 17: отражение

Examples

Надежное отражение с использованием кодов F

Отражение полезно, но хрупко. Учти это:

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

Проблемы с таким кодом:

1. Код не работает, потому что существует несколько перегрузок `String.StartsWith`
2. Даже если бы не было никаких перегрузок прямо сейчас, то более поздние версии библиотеки могут добавить перегрузку, приводящую к сбою во время выполнения
3. Инструменты рефакторинга, такие как `Rename methods` разбиты с отражением.

Это означает, что мы получаем сбои во время выполнения для того, что известно время компиляции. Это кажется субоптимальным.

Используя цитаты `F#` , можно избежать всех вышеперечисленных проблем. Определим некоторые вспомогательные функции:

```
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
```

Мы используем следующие функции:

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

Это печатает:

```
Boolean StartsWith(System.String)
Void .ctor(Char[])
```

```
Microsoft.FSharp.Collections.FSharpList`1[System.Int32] Singleton[Int32](Int32)
```

`<@ ... @>` означает, что вместо выполнения выражения внутри `F#` генерируется дерево выражений, представляющее выражение. `<@ "".StartsWith "" @>` генерирует дерево выражений, которое выглядит так: `Call (Some (Value ("")), StartsWith, [Value ("")])` . Это дерево выражений соответствует `getMethodInfo` и возвращает правильную информацию о методе.

Это устраняет все перечисленные выше проблемы.

Прочитайте отражение онлайн: https://riptutorial.com/ru/fsharp/topic/4124/отражение

глава 18: Параметры статического разрешения

Синтаксис

- `s` - это экземпляр `^a` вы хотите принять во время компиляции, что может быть тем, что реализует члены, которые вы на самом деле вызываете, используя синтаксис.
- `^a` аналогичен дженерикам, которые были бы `'a` (или `'A` или `'T` например), но они разрешены во время компиляции и допускают все, что соответствует всем запросам в методе. (не требуется никаких интерфейсов)

Examples

Простое использование для всего, что имеет член длины

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

Класс, интерфейс, использование записи

```
// 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)
```

Вызов статического участника

это будет принимать любой тип с методом `GetLength` который ничего не принимает и возвращает int:

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

Прочитайте Параметры статического разрешения онлайн: https://riptutorial.com/ru/fsharp/topic/7228/параметры-статического-разрешения

глава 19: Портирование C # на F

Examples

POCO которые

Некоторые из самых простых видов классов - POCOs.

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

В F # 3.0 были введены автоматические свойства, аналогичные авто свойствам C #,

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

Создание экземпляра либо похоже,

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

Если вы можете использовать неизменяемые значения, тип записи намного более идиоматичен F #.

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

И эта запись может быть создана:

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

Записи также могут быть созданы на основе других записей по specifiying существующей записи и добавления `with` , затем список полей для переопределения:

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

Класс Внедрение интерфейса

Классы реализуют интерфейс для соответствия контракту интерфейса. Например, класс C # может реализовать `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();
        }
    }
}
```

Чтобы реализовать интерфейс в F #, используйте `interface` в определении типа,

```
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 -> ()
```

Прочитайте Портирование C # на F # онлайн: https://riptutorial.com/ru/fsharp/topic/6828/портирование-c-sharp-на-f-sharp

глава 20: Последовательность

Examples

Генерировать последовательности

Существует несколько способов создания последовательности.

Вы можете использовать функции модуля 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
```

Вы также можете использовать выражение последовательности:

```
// 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 }
}
```

Введение в последовательности

Последовательность представляет собой ряд элементов, которые можно перечислить. Это псевдоним System.Collections.Generic.Inumerable и ленивый. Он хранит ряд элементов

одного типа (может быть любое значение или объект, даже другая последовательность). Функции от Seq.module могут использоваться для работы с ним.

Вот простой пример перечисления последовательности:

```
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
```

Выход:

```
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; ...]
```

Примените функцию к каждому элементу последовательности, используя Seq.map

Seq.filter

Предположим, что у нас есть последовательность целых чисел, и мы хотим создать последовательность, содержащую только четные целые числа. Мы можем получить последнее, используя функцию `filter` модуля Seq. Функция `filter` имеет сигнатуру типа `('a -> bool) -> seq<'a> -> seq<'a>` ; это означает, что он принимает функцию, которая возвращает true или false (иногда называемый предикатом) для заданного ввода типа `'a` и последовательности, которая содержит значения типа `'a` чтобы получить последовательность, которая содержит значения типа `'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
```

Бесконечные повторяющиеся последовательности

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

Повторяющиеся последовательности могут быть созданы с использованием выражения

вычисления `seq {}`

Прочитайте Последовательность онлайн: https://riptutorial.com/ru/fsharp/topic/2354/последовательность

глава 21: Последовательность рабочих процессов

Examples

урожайность и урожайность!

В последовательных рабочих процессах `yield` добавляет один элемент в построенную последовательность. (В монадической терминологии это `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!` (произносится как *ударный удар*) вставляет все элементы другой последовательности в эту построенную последовательность. Или, другими словами, он добавляет последовательность. (В отношении монад, оно `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']
```

Последовательности, созданные потоками последовательности, также ленивы, что означает, что элементы последовательности фактически не оцениваются до тех пор, пока они не понадобятся. Несколько способов принудительного использования элементов включают вызов `Seq.take` (вытягивает первые n элементов в последовательность), `Seq.iter` (применяет функцию к каждому элементу для выполнения побочных эффектов) или `Seq.toList` (преобразует последовательность в список) , Сочетание этого с рекурсией - это `yield!` действительно начинает светиться.

```
> 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]
```

за

`for` выражения последовательности предназначен для того, чтобы выглядеть так же, как его более известный двоюродный брат, императив для петли. Он «петляет» через последовательность и оценивает тело каждой итерации в последовательности, которую он генерирует. Как и вся связанная последовательность, она НЕ изменчива.

```
> 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]
```

Прочитайте Последовательность рабочих процессов онлайн:
https://riptutorial.com/ru/fsharp/topic/2785/последовательность-рабочих-процессов

глава 22: Процессор почтовых ящиков

замечания

`MailboxProcessor` поддерживает внутреннюю очередь сообщений, где несколько производителей могут отправлять сообщения с использованием различных вариантов метода `Post` . Эти сообщения затем извлекаются и обрабатываются одним потребителем (если вы не реализуете его в противном случае) с использованием вариантов `Retrieve` и `Scan` . По умолчанию, как производство, так и потребление сообщений являются потокобезопасными.

По умолчанию обработка ошибок отсутствует. Если в тело процессора выбрано неперехваченное исключение, функция body прекратится, все сообщения в очереди будут потеряны, больше сообщений не будет опубликовано, и канал ответа (если он доступен) получит исключение вместо ответа. Вы должны предоставить всю обработку ошибок самостоятельно, если это поведение не подходит для вашего использования.

Examples

Основной Hello World

Давайте сначала создадим простой «Hello world!». `MailboxProcessor` который обрабатывает сообщения одного типа и печатает приветствия.

Вам понадобится тип сообщения. Это может быть что угодно, но дискриминированные союзы являются естественным выбором здесь, поскольку они перечисляют все возможные случаи в одном месте, и вы можете легко использовать сопоставление образцов при их обработке.

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

Теперь определите сам процессор. Это можно сделать с помощью `MailboxProcessor<'message>.Start` Статический метод, который возвращает запущенный процессор, готовый выполнить свою работу. Вы также можете использовать конструктор, но тогда вам нужно обязательно запустить процессор позже.

```
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 ())
```

Параметр `Start` - это функция, которая ссылается на сам `MailboxProcessor` (который еще не существует, поскольку вы только создаете его, но будет доступен после выполнения функции). Это дает вам доступ к различным методам `Receive` и `Scan` для доступа к сообщениям из почтового ящика. Внутри этой функции вы можете делать любую необходимую обработку, но обычный подход - это бесконечный цикл, который читает сообщения один за другим и вызывает себя после каждого.

Теперь процессор готов, но это ни к чему! Зачем? Вам нужно отправить сообщение для обработки. Это делается с вариантами метода `Post` - давайте используем самый простой, с легкостью и забытый.

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

Это помещает сообщение во внутреннюю очередь `processor` , почтовый ящик и немедленно возвращается, чтобы код вызова мог продолжаться. Как только процессор получит сообщение, он обработает его, но это будет сделано асинхронно для его публикации и, скорее всего, будет выполнено в отдельном потоке.

Очень скоро после этого вы увидите сообщение `"Hi, Alice! This is mailbox processor's inner loop!"` напечатаны на выходе, и вы готовы к более сложным образцам.

Управление взаимным управлением

Процессоры почтовых ящиков могут использоваться для управления изменяемым состоянием прозрачным и потокобезопасным способом. Давайте построим простой счетчик.

```
// 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
```

Теперь давайте сгенерируем некоторые операции

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

И вы увидите следующий журнал

```
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
```

совпадение

Поскольку процессор почтовых ящиков обрабатывает сообщения один за другим, и нет чередования, вы также можете создавать сообщения из нескольких потоков, и вы не увидите типичных проблем с потерянными или дублируемыми операциями. Невозможно, чтобы сообщение использовало старое состояние других сообщений, если вы специально не применяете процессор.

```
let processor = createProcessor 0

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

Все сообщения публикуются из разных тем. Порядок, в котором сообщения отправляются в

почтовый ящик, не является детерминированным, поэтому порядок их обработки не является детерминированным, но поскольку общее количество приращений и декрементов сбалансировано, вы увидите, что конечное состояние равно 0, независимо от того, в каком порядке и из каких потоков были отправлены сообщения.

Истинное изменяемое состояние

В предыдущем примере мы только моделировали изменяемое состояние, передавая параметр рекурсивного цикла, но процессор почтовых ящиков обладает всеми этими свойствами даже для действительно изменяемого состояния. Это важно, когда вы поддерживаете большое состояние и неизменность, непрактично по соображениям производительности.

Мы можем переписать наш счетчик на следующую реализацию

```
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 ())
```

Хотя это определенно не было бы потокобезопасным, если состояние счетчика было изменено непосредственно из нескольких потоков, вы можете увидеть, используя параллельное сообщение Сообщения из предыдущего раздела, что процессор почтовых ящиков обрабатывает сообщения один за другим без перемежения, поэтому каждое сообщение использует самое текущее значение.

Возвращаемые значения

Вы можете асинхронно возвращать значение для каждого обработанного сообщения, если вы отправляете `AsyncReplyChannel<'a>` как часть сообщения.

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

Затем процессор почтовых ящиков может использовать этот канал при обработке сообщения для отправки значения обратно вызывающему абоненту.

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

Теперь, чтобы создать сообщение, вам нужен `AsyncReplyChannel<'a>` - что есть и как вы создаете рабочий экземпляр? Лучший способ - позволить MailboxProcessor предоставить его для вас и извлечь ответ на более распространенный `Async<'a>` . Это можно сделать, используя, например, метод `PostAndAsynReply` , в котором вы не `PostAndAsynReply` полное сообщение, а вместо этого - функцию типа (в нашем случае) `AsyncReplyChannel<OutputData> -> MessageWithResponse` :

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

Это отправит сообщение в очередь и ждет ответа, который поступит после того, как процессор получит это сообщение и ответит с использованием канала.

Существует также синхронный вариант `PostAndReply` который блокирует вызывающий поток, пока процессор не ответит.

Обработка сообщений вне ордера

Вы можете использовать методы `Scan` или `TryScan` для поиска определенных сообщений в очереди и обработки их независимо от количества сообщений перед ними. Оба метода просматривают сообщения в очереди в том порядке, в котором они прибыли, и будут искать указанное сообщение (вплоть до дополнительного тайм-аута). Если такого сообщения нет, `TryScan` вернет None, в то время как `Scan` продолжит ждать, пока не поступит такое сообщение или не `TryScan` время работы.

Давайте посмотрим это на практике. Мы хотим, чтобы процессор обрабатывал `RegularOperations` когда это возможно, но всякий раз, когда есть `PriorityOperation` , его следует обрабатывать как можно скорее, независимо от того, сколько других `RegularOperations` находится в очереди.

```
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())
```

Прочитайте Процессор почтовых ящиков онлайн: https://riptutorial.com/ru/fsharp/topic/9409/процессор-почтовых-ящиков

глава 23: Расширения типа и модуля

замечания

Во всех случаях при расширении типов и модулей расширяющийся код должен быть добавлен / загружен перед кодом, который должен его вызвать. Он также должен быть доступен для вызывающего кода путем открытия / импорта соответствующих пространств имен.

Examples

Добавление новых методов / свойств в существующие типы

F # позволяет добавлять функции как «члены» к типам, когда они определены (например, типы записей). Однако F # также позволяет добавлять новые члены экземпляра в *существующие* типы - даже те, которые объявлены в другом месте и на других языках .net.

Следующий пример добавляет новый метод экземпляра. `Duplicate` все экземпляры `String` .

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

Примечание : `this` произвольно выбранное имя переменной, которое будет использоваться для ссылки на экземпляр расширяемого типа - `x` будет работать так же хорошо, но, возможно, будет менее самоописательным.

Затем он может быть вызван следующими способами.

```
// 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
```

Эта функциональность очень похожа на методы расширения в C #.

Аналогичным образом к существующим типам можно добавлять новые свойства. Они автоматически станут свойствами, если новый член не принимает аргументов.

```
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
```

Добавление новых статических функций в существующие типы

F # позволяют расширить существующие типы с помощью новых статических функций.

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

Эта новая функция может быть вызвана следующим образом:

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

Эта функция может означать, что вместо того, чтобы создавать «полезные» библиотеки функций, они могут быть добавлены к соответствующим существующим типам. Это может быть полезно для создания более функциональных версий функций F #, которые позволяют использовать такие функции, как currying .

```
type System.String with
    static member AreEqual comparer a b = System.String.Equals(a, b, comparer)

let caseInsensitiveEquals = String.AreEqual StringComparison.OrdinalIgnoreCase

let result = caseInsensitiveEquals "abc" "aBc"
// result is True
```

Добавление новых функций в существующие модули и типы с использованием модулей

Модули могут использоваться для добавления новых функций в существующие модули и типы.

```
namespace FSharp.Collections

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

Новая функция может быть вызвана так, как если бы она была оригинальным членом List.

```
open FSharp.Collections

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

Прочитайте Расширения типа и модуля онлайн: https://riptutorial.com/ru/fsharp/topic/2977/расширения-типа-и-модуля

глава 24: Реализация шаблона проектирования в F

Examples

Программирование, управляемое данными в F

Благодаря методу ввода-вывода и частичного применения в программировании, управляемом данными `F#` является лаконичным и читаемым.

Представим себе, что мы продаем автострахование. Прежде чем мы попытаемся продать его клиенту, мы попытаемся определить, является ли клиент действительным потенциальным клиентом для нашей компании, проверяя пол и возраст клиента.

Простая модель клиента:

```
type Sex =
  | Male
  | Female

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

Затем мы хотим определить список исключений (таблицу), чтобы, если клиент соответствует любой строке в списке исключений, клиент не может купить наше страхование автомобиля.

```
//  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
  |]
```

Из-за типового вывода и частичного приложения список исключений является гибким, но понятным.

Наконец, мы определяем функцию, которая использует список исключений (таблицу) для

разделения клиентов на два ведра: потенциальные и отрицаемые клиенты.

```
// 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 ()
```

Чтобы обернуть, давайте определим некоторых клиентов и посмотрим, являются ли они потенциальными клиентами для нашего страхования автомобилей среди них:

```
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
```

Это печатает:

```
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;})|]
```

Прочитайте Реализация шаблона проектирования в F # онлайн: https://riptutorial.com/ru/fsharp/topic/3925/реализация-шаблона-проектирования-в-f-sharp

глава 25: Складки

Examples

Введение в складки, с несколькими примерами

Складки - это функции (более высокого порядка), используемые с последовательностями элементов. Они сворачивают `seq<'a>` в `'b` где `'b` - любой тип (возможно, еще `'a`). Это немного абстрактно, поэтому давайте рассмотрим конкретные практические примеры.

Вычисление суммы всех чисел

В этом примере `'a` является `int` . У нас есть список чисел, и мы хотим рассчитать сумму всех его чисел. Суммировать номера списка `[1; 2; 3]` пишем

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

Позвольте мне объяснить, потому что мы имеем дело со списками, мы используем `fold` в модуле `List` , следовательно `List.fold` . первый аргумент `fold` принимает двоичную функцию - **папку** . Второй аргумент - это **начальное значение** . `fold` начинает складывать список, последовательно применяя функцию папки к элементам в списке, начиная с начального значения и первого элемента. Если список пуст, возвращается исходное значение!

Схематический пример примера выполнения выглядит так:

```
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
```

Функция `List.sum` примерно `List.fold add LanguagePrimitives.GenericZero` где общий ноль делает его совместимым с целыми числами, поплавками, большими целыми числами и т. Д.

Подсчет элементов в списке (`count`

выполнения)

Это делается почти так же, как указано выше, но игнорируя фактическое значение элемента в списке и вместо этого добавляя 1.

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

Это также можно сделать следующим образом:

```
[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
```

Таким образом, вы можете определить `count` следующим образом:

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

Поиск максимального списка

На этот раз мы будем использовать `List.reduce` который похож на `List.fold` но без начального значения, как в этом случае, когда мы не знаем, что тип имеет значения, которые мы компилируем:

```
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
```

Поиск минимума списка

Точно так же, как при поиске max, папка отличается

```
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
```

Конкатенационные списки

Здесь мы берем список списков. Функция папки - это оператор `@`

```
// [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]
```

Или вы можете использовать двоичные операторы в качестве функции вашей папки:

```
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...
```

Вычисление факториала числа

Та же идея, что и при суммировании чисел, но теперь мы их умножаем. если мы хотим, чтобы факториал `n` умножал все элементы в списке `[1 .. n]` . Код становится:

```
// 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]
```

Реализация `forall` `exists` и `contains`

функция `forall` проверяет, удовлетворяют ли все элементы в последовательности условию. `exists` проверка, если по крайней мере один элемент в списке удовлетворяет условию. Сначала нам нужно знать, как свернуть список значений `bool` . Ну, мы используем складки конечно! Булевыми операторами будут наши функции папок.

Чтобы проверить, `true` ли все элементы в списке, мы сбрасываем их с помощью функции `&&` с `true` значением в качестве начального значения.

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

Аналогично, чтобы проверить, является ли один элемент `true` в логическом списке, мы свернем его с `||` оператор с `false` как начальное значение:

```
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
```

Вернуться к `forall` и `exists` . Здесь мы берем список любого типа, используем условие для преобразования всех элементов в логические значения, а затем мы сворачиваем его:

```
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
```

Проверить, все ли элементы в [1; 2; 3; 4] меньше 5:

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

определить метод `contains` with `exists` :

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

Или даже

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

Теперь проверьте, содержит ли список [1 .. 5] значение 2:

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

Внедрение `reverse` :

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

Реализация `map` и `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]
```

Есть ли что - нибудь `fold` не может сделать? Я не знаю

Вычисление суммы всех элементов списка

Чтобы вычислить сумму терминов (типа float, int или большое целое число) списка номеров, предпочтительнее использовать List.sum. В других случаях List.fold - это функция, которая лучше всего подходит для вычисления такой суммы.

1. Сумма комплексных чисел

В этом примере мы объявляем список комплексных чисел и вычисляем сумму всех членов в списке.

В начале программы добавьте ссылку на System.Numerics

открыть System.Numerics

Чтобы вычислить сумму, мы инициализируем аккумулятор до комплексного числа 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
```

Результат:

```
(3, 51)
```

2. Сумма чисел типа объединения

Предположим, что список состоит из чисел union (float или int) и хочет вычислить сумму этих чисел.

Объявите перед следующим типом номера:

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

Вычислить сумму номеров типа списка:

```
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
```

Результат:

```
13.5
```

Первый параметр функции, который представляет собой накопитель, имеет тип float, а второй параметр, который представляет элемент в списке, имеет номер типа. Но перед тем, как мы добавим, нам нужно использовать сопоставление образцов и приведение к типу float, когда elem имеет тип Int.

Прочитайте Складки онлайн: https://riptutorial.com/ru/fsharp/topic/2250/складки

глава 26: Соответствие шаблону

замечания

Совместимость шаблонов - это мощная функция многих функциональных языков, поскольку она часто позволяет обрабатывать ветвление очень кратко по сравнению с использованием нескольких операторов `if` / `else if` / `else` . Однако, учитывая достаточно вариантов и «когда» охранники , сопоставление образцов также может стать многословным и трудно понять с первого взгляда.

Когда это происходит, активные шаблоны F # могут стать отличным способом дать значимые имена логике соответствия, что упрощает код, а также позволяет повторно использовать.

Examples

Варианты соответствия

Согласование шаблонов может быть полезно для обработки параметров:

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

Проверка соответствия шаблонов всему домену

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

дает предупреждение

> C: \ Program Files (x86) \ Microsoft VS Code \ Untitled-1 (2,7): предупреждение FS0025: Неполные совпадения шаблонов в этом выражении. Например, значение «false» может указывать на случай, не охваченный шаблоном (-ами).

Это связано с тем, что не все возможные значения bool были покрыты.

bools могут быть явно указаны, но ints

сложнее перечислять

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

здесь мы используем специальный символ _ . _ Соответствует всем другим возможным случаям.

_ Может вызвать у вас неприятности

рассмотрим тип, который мы создаем, он выглядит так

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

Мы могли бы написать матч с выпиской, который выглядит так

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

Вышеприведенный код имеет смысл. Мы предполагаем, что если вы не трезвы, вам следует назвать uber, поэтому мы используем _ чтобы обозначить, что

Позднее мы реорганизуем наш код на этот

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

Компилятор F # должен дать нам предупреждение и пригласить нас реорганизовать наше выражение соответствия, чтобы человек обратился за медицинской помощью. Вместо этого выражение матча бесшумно относится к бессознательному человеку, как если бы они были только подвыпившими. Дело в том, что вы должны выбрать, чтобы в явном виде перечислять случаи, когда это возможно, чтобы избежать логических ошибок.

Случаи оцениваются сверху донизу и используется первое совпадение

Неправильное использование:

В следующем фрагменте последний матч никогда не будет использоваться:

```
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"
```

печать

> x - это то, что не было указано выше

Правильное использование:

Здесь как `x = 1` и `x = 4` попадут в их конкретные случаи, а все остальное попадет в случай по умолчанию _ :

```
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"
```

печать

> x равно 4

Когда охранники позволяют добавлять произвольные условные обозначения

```
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"
```

Прочитайте Соответствие шаблону онлайн: https://riptutorial.com/ru/fsharp/topic/1335/соответствие-шаблону

глава 27: Списки

Синтаксис

- [] // пустой список.

  head :: tail // строительная ячейка, содержащая элемент, головку и список, хвост. :: называется оператором Cons.

  let list1 = [1; 2; 3] // Обратите внимание на использование точки с запятой.

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

  let list3 = list1 @ list2 // result is [1; 2; 3; 0; 1; 2; 3]. @ - оператор добавления.

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

  let list5 = [1..2..10] // результат - [1; 3; 5; 7; 9]

  пусть list6 = [для i в 1..10 do, если i% 2 = 1, тогда выведите i] // результат - [1; 3; 5; 7; 9]

Examples

Использование основного списка

```
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
```

Вычисление общей суммы чисел в списке

По рекурсии

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

В приведенном выше примере говорится: «Посмотрите на `list` , он пуст? Return 0. В противном случае это непустой список, поэтому он может быть `[1]` , `[1; 2]` , [1; 2; 3] и т. Д. Если `list` равен [1], тогда привяжите переменную `head` к `1` и `tail` к `[]` затем запустите `head + sumTotal tail` .

Пример выполнения:

```
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
```

Более общий способ инкапсуляции вышеуказанного шаблона - использование функциональных складок! `sumTotal` становится следующим:

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

Создание списков

Способ создания списка состоит в размещении элементов в двух квадратных скобках, разделенных точками с запятой. Элементы должны иметь один и тот же тип.

Пример:

```
> 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]
```

Если в списке нет элемента, он пуст. Пустой список можно объявить следующим образом:

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

Другой пример

Чтобы создать список байтов, просто выделите целые числа:

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

Также возможно определить списки функций, элементов определенного ранее типа, объектов класса и т. Д.

пример

```
> 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]
```

Изменяется

Для определенных типов элементов (int, float, char, ...), можно определить список с помощью элемента start и конечного элемента, используя следующий шаблон:

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

Примеры:

```
> 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]
```

Вы также можете указать шаг для определенных типов со следующей моделью:

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

Примеры:

```
> 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]
```

Генератор

Другим способом создания списка является его автоматическое создание с помощью генератора.

Мы можем использовать одну из следующих моделей:

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

или же

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

Примеры

```
> 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]
```

операторы

Некоторые операторы могут использоваться для построения списков:

Против оператора:

Этот оператор :: используется для добавления элемента заголовка в список:

```
> 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]
```

конкатенация

Конкатенация списков выполняется с помощью оператора @.

```
> 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]
```

Прочитайте Списки онлайн: https://riptutorial.com/ru/fsharp/topic/1268/списки

глава 28: Струны

Examples

Строковые литералы

```
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
```

Простое форматирование строк

Существует несколько способов форматирования и получения строки.

.NET способом является использование `String.Format` или `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 # также имеет функции форматирования строки в стиле C. Для каждой функции .NET существуют эквиваленты:

- `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` (`bprintf`):

```
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
```

Использование этих функций вместо функций .NET дает некоторые преимущества:

- Тип безопасности
- Частичное применение
- Поддержка родного типа F #

Прочитайте Струны онлайн: https://riptutorial.com/ru/fsharp/topic/1397/струны

глава 29: Тип «единицы»

Examples

Что хорошего в 0-кортеже?

Набор из 2-х кортежей или 3-кортежей представляет собой группу связанных предметов. (Точки в 2D-пространстве, RGB-значения цвета и т. Д.). 1-кортеж не очень полезен, поскольку его можно легко заменить одним `int` .

0-кортеж кажется еще более бесполезным, поскольку в нем *нет* абсолютно *ничего* . Тем не менее он обладает свойствами, которые делают его очень полезным в функциональных языках, таких как F #. Например, тип 0-кортежа имеет ровно *одно* значение, обычно представленное как `()` . Все 0-кортежи имеют это значение, поэтому он по существу является одноэлементным. В большинстве функциональных языков программирования, включая F #, это называется типом `unit` .

Функции, возвращающие `void` в C #, возвращают тип `unit` в F #:

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

Запустите это в интерактивном интерпретаторе F #, и вы увидите:

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

Это означает, что значение `printResult` имеет тип `unit` и имеет значение `()` (пустой кортеж, одно и единственное значение типа `unit`).

Функции также могут принимать тип `unit` в качестве параметра. В F # функции могут выглядеть так, как будто они не принимают никаких параметров. Но на самом деле они берут единственный параметр `unit` типа. Эта функция:

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

на самом деле эквивалентно:

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

То есть, функция, которая принимает один параметр `unit` типа и возвращает значение `int` 6. Если вы посмотрите на подпись типа, которую интерпретирует интерактивный интерпретатор F # при определении этой функции, вы увидите:

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

Тот факт, что все функции будут принимать по крайней мере один параметр и возвращать значение, даже если это значение иногда является «бесполезным» значением like `()` , означает, что композиция функций намного проще в F #, чем на языках, на которых нет тип `unit` . Но это более продвинутый предмет, о котором мы поговорим позже. Пока просто помните, что когда вы видите `unit` в сигнатуре функции или `()` в параметрах функции, это тип 0-кортежа, который служит в качестве способа сказать: «Эта функция принимает или возвращает значения, значимые».

Отложить выполнение кода

Мы можем использовать тип `unit` в качестве аргумента функции для определения функций, которые мы не хотим выполнять до конца. Это часто полезно в асинхронных фоновых задачах, когда основной поток может запускать некоторые предопределенные функции фонового потока, например, может быть, переместить его в новый файл или если аа let-binding не следует запускать сразу:

```
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)
```

В следующем коде мы определяем код для запуска «рабочего», который просто выводит значение, которое он работает каждые 2 секунды. Затем рабочий возвращает две функции, которые могут использоваться для управления им - одна, которая перемещает ее до следующего значения для работы, и которое перестает работать. Они должны быть функциями, потому что мы не хотим, чтобы их тела исполнялись до тех пор, пока мы не примем решение, иначе рабочий немедленно переместится ко второму значению и выключится, не сделав ничего.

```
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
```

Затем мы можем начать работника,

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

и работа начнется - если мы находимся в интерактивном режиме F #, мы увидим, что сообщения печатаются на консоли каждые две секунды. Затем мы можем запустить

```
nextValue ()
```

и мы увидим сообщения, указывающие, что обрабатываемая ценность переместилась на следующую.

Когда пришло время закончить работу, мы можем запустить

```
stopOnNextTick ()
```

функция, которая распечатает закрывающее сообщение, затем выйдет.

`unit` типа важно здесь не означает «нет» вход - функции уже есть вся информация , они должны работать встроенные в них, и вызывающий абонент не может изменить.

Прочитайте Тип «единицы» онлайн: https://riptutorial.com/ru/fsharp/topic/2513/тип--единицы-

глава 30: Тип Провайдеры

Examples

Использование поставщика CSV-типа

Учитывая следующий файл CSV:

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

Вы можете прочитать данные со следующим скриптом:

```
#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
```

Использование поставщика типа WMI

Поставщик типа WMI позволяет запрашивать службы WMI с сильной типизацией.

Чтобы вывести результаты запроса WMI как 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)
```

Прочитайте Тип Провайдеры онлайн: https://riptutorial.com/ru/fsharp/topic/1631/тип-провайдеры

глава 31: Типы

Examples

Введение в типы

Типы могут представлять разные вещи. Это могут быть отдельные данные, набор данных или функция.

В F # мы можем сгруппировать типы по двум категориям:

- Типы 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
```

- Типы .NET
  - Встроенный тип (int, bool, string, ...)
  - Классы, структуры и интерфейсы
  - Делегаты
  - Массивы

Аббревиатуры типов

Аббревиатуры типов позволяют создавать псевдонимы на существующих типах, чтобы придать им более значимые ощущения.

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

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

Затем вы можете использовать псевдоним так же, как и любой другой тип:

```
// 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";}
```

Будьте осторожны, алиасы не проверяют соответствие типов. Это означает, что два псевдонима, которые нацелены на один и тот же тип, могут быть назначены друг другу:

```
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";}
```

Типы создаются в F #, используя ключевое слово типа

`F#` использует ключевое слово `type` для создания различных типов типов.

1. Тип псевдонимов
2. Типы дискриминационного союза
3. Типы записей
4. Типы интерфейсов
5. Типы классов
6. Типы типов

Примеры с эквивалентным кодом `C#` где это возможно:

```
// 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
```

Вывод типа

Подтверждение

Этот пример адаптирован из этой статьи о типе вывода

Что такое вывод типа?

Type Inference - это механизм, позволяющий компилятору определить, какие типы используются и где. Этот механизм основан на алгоритме, который часто называют « Hindley-Milner» или «HM». Ниже приведены некоторые правила определения типов простых и функциональных значений:

- Посмотрите на литералы
- Посмотрите на функции и другие ценности, с которыми что-то взаимодействует
- Посмотрите на любые явные ограничения типа
- Если в любом месте нет ограничений, автоматически обобщайте общие типы

Посмотрите на литералы

Компилятор может выводить типы, просматривая литералы. Если литерал является int, и вы добавляете к нему «x», тогда «x» также должен быть int. Но если литерал является float, и вы добавляете к нему «x», тогда «x» также должен быть float.

Вот некоторые примеры:

```
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"
```

Посмотрите на функции и другие значения, с которыми он взаимодействует

Если литералов нет, компилятор пытается выработать типы, анализируя функции и другие значения, с которыми они взаимодействуют.

```
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
```

Посмотрите на какие-либо явные ограничения типа или аннотации

Если есть какие-либо явные ограничения типа или аннотации, то компилятор будет их использовать.

```
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
```

Автоматическое обобщение

Если после всего этого не обнаружено ограничений, компилятор просто делает типы обобщенными.

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

Вещи, которые могут пойти не так с типом вывода

Увы, тип вывода не увенчан успехом. Иногда компилятор просто не знает, что делать. Опять же, понимание того, что происходит, действительно поможет вам оставаться спокойным, а не желать убить компилятора. Вот некоторые из основных причин ошибок типа:

- Объявления не в порядке

- Не хватает информации
- Перегруженные методы

Объявления не в порядке

Основное правило заключается в том, что вы должны объявлять функции до их использования.

Этот код не работает:

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

Но это нормально:

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

Рекурсивные или одновременные объявления

Вариант проблемы «не в порядке» возникает с рекурсивными функциями или определениями, которые должны ссылаться друг на друга. Никакое количество переупорядочения не поможет в этом случае - нам нужно использовать дополнительные ключевые слова, чтобы помочь компилятору.

Когда функция компилируется, идентификатор функции недоступен для тела. Поэтому, если вы определите простую рекурсивную функцию, вы получите ошибку компилятора. Исправление состоит в том, чтобы добавить ключевое слово «rec» как часть определения функции. Например:

```
// 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
```

Вот фиксированная версия с добавлением «rec fib», чтобы указать, что она рекурсивна:

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

Не хватает информации

Иногда для определения типа компилятор просто не располагает достаточной информацией. В следующем примере компилятор не знает, на какой тип должен работать метод Length. Но он не может сделать его общим, поэтому он жалуется.

```
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 ...
```

Эти ошибки могут быть исправлены с помощью явных аннотаций.

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

Перегруженные методы

При вызове внешнего класса или метода в .NET вы часто будете получать ошибки из-за перегрузки.

Во многих случаях, таких как пример concat ниже, вам придется явно аннотировать параметры внешней функции, чтобы компилятор знал, какой перегруженный метод нужно вызвать.

```
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
```

Иногда перегруженные методы имеют разные имена аргументов, и в этом случае вы также можете дать компилятору ключ, называя аргументы. Ниже приведен пример конструктора StreamReader.

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

Прочитайте Типы онлайн: https://riptutorial.com/ru/fsharp/topic/3559/типы

глава 32: Типы опций

Examples

Определение варианта

`Option` - это дискриминационный союз с двумя случаями: « `None` или « `Some` .

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

Используйте опцию <'T> над нулевыми значениями

В языках функционального программирования, таких как значения `F#` `null` , считается потенциально опасным и плохим стилем (неидиоматическим).

Рассмотрим этот код `C#` :

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

`x.Length` будет бросать, если `x` равно `null` , добавим защиту:

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

Или же:

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

Или же:

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

В идиоматических значениях `F#` `null` не используются, поэтому наш код выглядит так:

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

Однако иногда возникает необходимость в представлении пустых или недопустимых значений. Затем мы можем использовать `Option<'T>` :

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

`SomeFunction` либо возвращает `Some string` значение, либо `None` . Мы извлекаем `string` значение с использованием соответствия шаблону

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

Причина этого кода менее хрупкая:

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

Это потому, что мы не можем вызывать `Length` по `string option` . Нам нужно извлечь `string` значение с помощью сопоставления шаблонов, и тем самым мы гарантируем, что `string` значение будет безопасно использовать.

Опциональный модуль позволяет навигация по железной дороге

Обработка ошибок важна, но может превратить изящный алгоритм в беспорядок. Жестко ориентированное программирование (`ROP`) используется для упрощения обработки ошибок и создания композиций.

Рассмотрим простую функцию `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
```

Целью `f` является для разбора входной `string` значения (если есть `Some`) в `int` . Если `int` больше `0` мы превращаем его в `float` . Во всех остальных случаях мы спасаемся с `None` .

Несмотря на то, что чрезвычайно простая функция вложенного `match` значительно уменьшает читаемость.

`ROP` отмечает, что у нас есть два пути выполнения в нашей программе

1. Счастливый путь - в конечном итоге вычислит `Some` значение
2. Путь ошибок - все остальные пути не `None`

Поскольку пути ошибок более часты, они, как правило, используют код. Нам бы хотелось,

чтобы код счастливого пути был наиболее видимым кодом.

Эквивалентная функция `g` использующая `ROP` может выглядеть так:

```
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
```

Это очень похоже на то, как мы склонны обрабатывать списки и последовательности в `F#` .

Вы можете увидеть `Option<'T>` как `List<'T>` который может содержать только `0` или `1` элемент, где `Option.bind` ведет себя как `List.pick` (концептуально `Option.bind` лучше отображает `List.collect` но `List.pick` может быть легче понять).

`bind` , `filter` и `map` обрабатывает пути ошибок, а `g` содержит только код счастливого пути.

Все функции, которые непосредственно принимают `Option<_>` и возвращают `Option<_>` , напрямую могут быть скомпилированы с помощью `|>` и `>>` .

Таким образом, `ROP` повышает читаемость и способность к сшиванию.

Использование типов опций из C

Не рекомендуется выводить типы опций на код C #, так как у C # нет способа справиться с ними. Варианты - либо ввести `FSharp.Core` как зависимость в вашем проекте C # (что вам нужно сделать, если вы потребляете библиотеку F #, *не* предназначенную для взаимодействия с C #), либо изменить значения `None` на `null` .

Pre-F # 4.0

Способ сделать это - создать собственную функцию преобразования:

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

Для типов значений вам придется прибегать к боксу или использовать `System.Nullable` .

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

F # 4.0

В F # 4.0 функции `ofObj` , `toObj` , `ofNullable` и `toNullable` если они введены в модуль `Option` . В F # interactive они могут использоваться следующим образом:

```
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
```

Обратите внимание, что `None` компилируется с `null` внутренним. Однако, что касается F #, это `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
```

Прочитайте Типы опций онлайн: https://riptutorial.com/ru/fsharp/topic/3175/типы-опций

глава 33: функции

Examples

Функции более одного параметра

В F # **все функции принимают ровно один параметр** . Это кажется нечетным утверждением, поскольку тривиально легко объявить более одного параметра в объявлении функции:

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

Но если вы введете это объявление функции в интерактивный интерпретатор F #, вы увидите, что его подпись типа:

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

Без имен параметров эта подпись является `int -> int -> int` . Оператор `->` является право-ассоциативным, что означает, что эта сигнатура эквивалентна `int -> (int -> int)` . Другими словами, `add` - это функция, которая принимает один параметр `int` и возвращает **функцию, которая принимает один `int` и возвращает `int`** . Попытайся:

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

Тем не менее, вы также можете вызвать функцию, например, `add` более «обычную» манеру, прямо передав ей два параметра, и она будет работать так, как вы ожидали:

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

Это относится к функциям с таким количеством параметров, которое вы хотите:

```
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
```

Этот метод мышления о многопараметрических функциях как функции, которые принимают один параметр и возвращают новые функции (которые, в свою очередь, могут принимать один параметр и возвращать новые функции, пока вы не достигнете конечной функции, которая принимает окончательный параметр и, наконец, возвращает результат) называется **currying** , в честь математика Haskell Curry, который славится разработкой концепции. (Он был изобретен кем-то другим, но Карри заслуженно получает большую часть кредита.)

Эта концепция используется во всем F #, и вы захотите ознакомиться с ней.

Основы функций

Большинство функций в F # создаются с помощью синтаксиса `let` :

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

Это определяет функцию с именем `timesTwo` которая принимает один параметр `x` . Если вы запускаете интерактивный сеанс F # (`fsharpi` на OS X и Linux, `fsi.exe` в Windows) и вставляете эту функцию в (и добавляете `;;` который сообщает `fsharpi` о том, что вы оцениваете только что введенный код), вы увидите, что это отвечает:

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

Это означает, что `timesTwo` - это функция, которая принимает один параметр `x` типа `int` и возвращает `int` . Сигнатуры функций часто записываются без имен параметров, поэтому вы часто увидите этот тип функции, записанный как `int -> int` .

Но ждать! Как F # знал, что `x` является целым параметром, так как вы никогда не указали его тип? Это связано с **типом вывода** . Поскольку в теле функции вы умножаете `x` на `2` , типы `x` и `2` должны быть одинаковыми. (Как правило, F # не будет неявно передавать значения для разных типов, вы должны явно указывать любые типы приемов).

Если вы хотите создать функцию, которая не принимает никаких параметров, это **неправильный** способ сделать это:

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

Правильный способ сделать это:

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

Эта функция `hello` имеет `unit -> unit` , который объясняется в типе «unit» .

Curried vs Tupled Functions

Существует два способа определения функций с несколькими параметрами в функциях F #, Curried и 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
```

Все функции, определенные извне F # (например, .NET framework), используются в F # с формой с расширением. Большинство функций в базовых модулях F # используются с формой Curried.

Форма Curried считается идиоматической F #, поскольку она допускает частичное применение. Ни один из следующих двух примеров невозможен с помощью формы «Тупик».

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

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

Причина этого в том, что функция Curried при вызове с одним параметром возвращает функцию. Добро пожаловать в функциональное программирование!

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

Вы можете видеть, что это точно такая же подпись.

Тем не менее, при взаимодействии с другим кодом .NET, как и при написании библиотек, важно определить функции, используя форму Tupled.

Встраивание

Вложение позволяет заменить вызов функции с телом функции.

Это иногда полезно для повышения производительности в критической части кода. Но аналогию является то, что ваша сборка займет много места, так как тело функции дублируется повсюду, когда произошел звонок. Вы должны быть осторожны, когда решаете, следует ли встраивать функцию или нет.

Функция может быть встроена в ключевое слово `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"
```

Другой пример с локальным значением:

```
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
```

Труба вперед и назад

Операторы труб используются для простого и элегантного передачи параметров функции. Это позволяет исключить промежуточные значения и упростить чтение функций.

В F # существуют два оператора:

- **Вперед** (`|>`): передача параметров слева направо

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

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

- **Назад** (`<|`): передача параметров справа налево

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

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

Вот пример без операторов:

```
// 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
```

Мы можем сократить предыдущий пример и сделать его более чистым с помощью оператора прямой трубы:

```
// 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
```

Каждый результат функции передается как параметр следующей функции.

Если вы хотите передать несколько параметров оператору трубы, вы должны добавить `|` для каждого дополнительного параметра и создать Кортеж с параметрами. Оператор прямой F # поддерживает до трех параметров (|||> или <|||).

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

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

Прочитайте функции онлайн: https://riptutorial.com/ru/fsharp/topic/2525/функции

кредиты

S. No	Главы	Contributors
1	Начало работы с F #	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	1: F # Код WPF для приложения с FsXaml	Bent Tranberg
3	F # на .NET Core	Boggin, Joel Martinez
4	F # Советы и подсказки производительности	FuleSnabel, Paul Westcott, Ringil, s952163
5	Активные шаблоны	Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
6	Введение в WPF в F #	Funk
7	Дженерики	Jake Lishman
8	Дискриминационные союзы	chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen , Vandroiy, VillasV
9	документация	eirik, goric, Ringil
10	Единицы измерения	asibahi, goric, GregC, Vandroiy
11	Использование F #, WPF, FsXaml, меню и диалогового окна	Bob McCrory, Goswin
12	Классы	asibahi, inzi, RamenChef, Tomasz Maczyński
13	Ленивая оценка	inzi
14	мемоизации	Jean-Claude Colette, Julien Pires, Ringil
15	Монады	FuleSnabel

16	операторы	FuleSnabel
17	отражение	FuleSnabel
18	Параметры статического разрешения	Maslow
19	Портирование C # на F #	jdphenix, marklam, RamenChef
20	Последовательность	Foggy Finder, inzi, James McCalden, Julien Pires, s952163
21	Последовательность рабочих процессов	Jwosty
22	Процессор почтовых ящиков	Honza Brestan
23	Расширения типа и модуля	Jono Job
24	Реализация шаблона проектирования в F #	FuleSnabel, Ringil
25	Складки	Jean-Claude Colette, Zaid Ajaj
26	Соответствие шаблону	asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene
27	Списки	asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj
28	Струны	FireAlkazar, Julien Pires
29	Тип «единицы»	4444, Abel, Jake Lishman, rmunn
30	Тип Провайдеры	GregC, jdphenix, Joel Martinez
31	Типы	Cedric Royer-Bertrand, FuleSnabel, Julien Pires
32	Типы опций	asibahi, chillitom, FuleSnabel
33	функции	asibahi, Julien Pires, rmunn, ronilk