Лекция 4: Типы и генерики в F#

Юрий Литвинов
y.litvinov@spbu.ru

1. Введение, генерики в F#

Как и все нормальные языки, F# поддерживает генерики, причём, поскольку F# работает на платформе .NET, то и генерики внутри реализованы так, как это делается на .NET. То есть так же, как в C#, есть открытые и закрытые типы[1], которые компилируются в соответствующие типы в IL. Так что, в отличии, например, от Java, байт-код всегда знает про тип, что он генерик, знает его параметры-типы (формальные и фактические), к ним можно получить доступ рефлексией и (в отличие от темплейтов C++) для инстанцирования генерика не требуются исходники.

Описываются генерики в F# двумя способами — в стиле OCaml (F# всё-таки создавался как диалект OCaml под .NET, так что требовалось поддерживать хотя бы до некоторой степени обратную совместимость), и в стиле C#. Вот объявление генерика в OCaml-стиле:

```
type 'a list = ...
```

А вот ~~в человеческом стиле~~ в стиле C#:

```
type Map<'TKey, 'TValue> = ...
```

В любом случае имена параметров-типов должны начинаться с одинарной кавычки, в остальном соглашения об именах параметров-типов такие же, как в C# ('T, если параметр-тип один, разумное имя, начинающееся с 'T, если несколько). OCaml-стиль позволяет указать только один параметр-тип, и там обычно используются сокращённые имена параметров-типов, например, 'a. В современном F# стиль с угловыми скобками считается предпочтительным.

Что более интересно, это то, что в F# используется *автоматическое обобщение*, то есть компилятор сам пытается сделать тип генериком, если это возможно, без явного указания параметров-типов со стороны программиста. Например, вы можете объявить функцию map вот таким образом:

```
let map = List.map
```

[1] Напомним, что открытым типом называется тип, имеющий формальные параметры-типы, на место которых не подставлены фактические, соответственно, создание объекта такого типа невозможно. Закрытые типы — типы, у которых каждому формальному параметру-типу поставлен в соответствие фактический тип, так что с ними можно работать как с негенериками. Например, `Dictionary<,>` — открытый тип, `Dictionary<int, string>` — закрытый.

И если исполнить эту невинно выглядящую строчку в F# Interactive, получим

```
val map: (('a -> 'b) -> 'a list -> 'b list)
```

— генерик с двумя параметрами-типами. Что логично, но никак не следует из её объявления. Мы могли бы объявить её генериком явно, вот так:

```
let map: ('a -> 'b) -> 'a list -> 'b list = List.map
```

Или даже вот так:

```
let map<'a, 'b> : ('a -> 'b) -> 'a list -> 'b list = List.map
```

Компилятор достаточно сообразителен, чтобы без перечисления параметров-типов в угловых скобках посмотреть на тип функции и понять, сколько каких параметров-типов она использует (в отличие от C#, кстати). Или самому вывести тип и сделать вывод, что функция может быть генериком. Поэтому все приведённые выше объявления функции map равнозначны.

Вообще, автоматическое обобщение призвано решить проблему с ленью программистов, которые часто не делают генериком то, что может быть генериком, а обходятся конкретными типами, чтобы не выписывать параметры-типы. Поскольку в F# параметры-типы выписывать и не нужно, пользоваться генериками в разы удобнее, поэтому код получается в среднем более переиспользуемым, чем, например, в C#. Причём выводятся даже довольно нетривиальные типы и отношения над типами:

```
let getFirst (a, b, c) = a
```

выведется в

```
val getFirst: 'a * 'b * 'c -> 'a
```

— компилятор догадался, что элементы тройки могут быть произвольного типа, но функция возвращает значение типа первого элемента тройки. Или ещё более хитрый пример:

```
let mapPair f g (x, y) = (f x, g y)
```

выведется в

```
val mapPair : ('a -> 'b) -> ('c -> 'd) -> ('a * 'c) -> ('b * 'd)
```

Тут компилятор справедливо заметит, что функции f и g могут действовать откуда угодно куда угодно, но пара-аргумент должна быть парой значений из областей определения функций, а пара-результат — соответственно из областей значений. Итого получилось четыре параметра-типа для генерик-функции mapPair.

2. Автоматический вывод типов

Всё это возможно благодаря автоматическому выводу типов, применяемому в F#, который тут уже неоднократно упоминался, но теперь наконец настала пора поговорить про него более подробно. В F# для вывода типов используется известный алгоритм Хиндли-Милнера, точнее, несколько модифицированный алгоритм «W» Дамаса-Милнера над системой типов Хиндли-Милнера. Система типов Хиндли-Милнера — это на самом деле одно из типизированных λ-исчислений, используется для вывода типов далеко не только в F# (например, Haskell работает концептуально похожим образом, хотя там меньше языкоспецифичных заморочек; и, разумеется, OCaml).

Интуитивно алгоритм работает, строя систему уравнений над типами. Изначально тип каждого выражения считается неизвестным и помечается «переменной типа», далее выводятся отношения над переменными типа. Например, если написано `let a = 1`, тип a изначально неизвестен, но let-определение накладывает ограничение, что тип a равен типу выражения после «=», а там стоит литерал типа int, тип которого точно известен, поэтому получили уравнение, решив которое получаем, что тип a равен int. В рассмотренном выше примере `let getFirst (a, b, c) = a` компилятор может построить только уравнение для первого элемента пары — что он равен типу возвращаемого значения функции.

Вообще в построении ограничений на типы используются литералы (в F#, как и в любом нормальном функциональном языке, типы литералов фиксированы и запрещено автоматическое приведение, что позволяет сразу зафиксировать типы в выражениях с литералами), функции и другие виды «взаимодействия значений» (операторы, например), явные ограничения на типы, аннотации типов (которые можно указать через двоеточие практически везде, где вводится какое-то имя).

Дальше уравнения решаются методом *унификации* — переменные типа постепенно исключаются, замещаясь в соответствии с известными ограничениями либо на конкретные типы, либо на другие переменные типа (либо выражения типа с этими переменными — составные типы, такие как списки, массивы, кортежи и т.д. — это суть операторы в «арифметике» типов, из которых можно составлять сколь угодно сложные выражения над типами). Как только никакую переменную типа исключить больше нельзя, процесс завершается, оставшиеся переменные типа становятся параметрами генериков (собственно, так и происходит автоматическое обобщение, это не то чтобы крутая фича компилятора, скорее побочный эффект алгоритма унификации).

Ещё важно понимать, что компилятор сначала составляет уравнения, а затем их решает, поэтому может учесть информацию о типах, доступную ниже в программе. На самом деле, вычисление типов глобально, то есть принимает во внимание все ограничения, про которые известно в момент компиляции, где бы в тексте программы те ни находились. Что хорошо, потому что (теоретически) можно о типах особо не заботиться, и если программа с точки зрения типов корректна, компилятор с ней разберётся (как мы увидим в дальнейшем, это не всегда правда, хоть чаще всего и работает). С другой стороны, могут быть и обратные эффекты — вы добавляете аннотацию типа где-то в глубине программы, перестаёт компилироваться какое-то совершенно другое место, потому что внезапно ограничения на типы становятся неразрешимы. Это часто вызывает огорчение, поэтому даже самые упёртые функциональные программисты стараются хотя бы в ключевых местах писать аннотации типов (в интерфейсах модулей, например).

Например, рассмотрим функцию

```
let outerFn action : string =
    let innerFn x = x + 1
    action (innerFn 2)
```

Известно, что функция возвращает значение типа string, но аргумент какого типа она принимает? Обозначим тип аргумента за 'a. Смотрим на `let innerFn x = x + 1` — тип x обозначим за 'b и вспомним, что оператор + имеет тип 'c -> 'c (это на самом деле не так, но достаточно близко к правде для этого примера), а 1 — это литерал типа int. Поэтому уравнение относительно 'b решается тривиально, тип x — int, тип innerFn — int -> int.

Теперь `action (innerFn 2)` говорит нам, что action — функция от одного аргумента, то есть 'a = 'd -> 'e, при этом 'e = string, поскольку он равен типу возвращаемого значения outerFn, а 'd = int, поскольку равен типу возвращаемого значения innerFn. И теперь мы наконец получили тип outerFn — (int -> string) -> string.

Рассмотрим также функцию

```
let doItTwice f = (f >> f)
```

doItTwice имеет тип 'a -> b, при этом оператор композиции имеет тип ('c -> 'd) -> ('d -> 'e) -> ('c -> 'e), и раз он получает f и первым, и вторым аргументом, то 'c = 'd и 'd = 'e, так что тип » в выражении (f » f) можно записать как ('c -> 'c) -> ('c -> 'c). При этом f у нас имеет тип 'a, так что 'a = 'c -> 'c. Итого получаем тип doItTwice: ('c -> 'c) -> ('c -> 'c), или, если переименовать переменные типа, ('a -> 'a) -> ('a -> 'a), то есть doItTwice — генерик, принимающий функцию, у которой множества определения и значений совпадают, и возвращающий такую же по типу функцию, которая дважды применяет исходную.

Подробное объяснение «на пальцах» можно посмотреть в https://fsharpforfunandprofit.com/posts/type-inference/ (оттуда же на самом деле позаимствованы примеры и общая идея изложения), а теория, стоящая за алгоритмом Хиндли-Милнера, хорошо изложена, как ни странно, в Википедии: https://en.wikipedia.org/wiki/Hindley–Milner_type_system

2.1. Особенности F#

Всё это хорошо работает в теории (и в Haskell), но на практике можно получить массу интересных ошибок компиляции. Например,

```
List.map (fun x -> x.Length) ["hello"; "world"]
```

не скомпилируется. В момент компиляции x.Length тип x неизвестен, поэтому доказать, что у него реально есть свойство Length, компилятор не может, соответственно не может и установить корректность вызова, следовательно, не даст такое скомпилить. Но ведь Хиндли-Милнер работает глобально, а у нас буквально сразу за этим выражением написано, что работаем мы со списком строк, из чего, зная тип List.map можно вывести, что x имеет тип string, так что выражение x.Length корректно? Да, но нет — ограничение на тип «'a такой, что у него есть свойство или метод (или поле) Length» в системе уравнений над типами не записать, так что запомнить использование .Length в лямбде компилятор не может, и тот факт, что x — это строка, ему не помогает, потому что у него нет повода

вернуться и проверить тип выражения ещё раз. Компилятор F# однопроходный, поэтому если он видел выражение, но никуда его не записал, то больше он его не увидит.

На самом деле, это ограничение системы вывода типов сильно мешает объектно-ориентированному программированию на F#, поскольку заставляет почти везде писать аннотации типов:

```
List.map (fun (x: string) -> x.Length) ["hello"; "world"]
```

— так скомпилируется вполне. Скомпилируется и вот так:

```
["hello"; "world"] |> List.map (fun x -> x.Length)
```

— тут мы к моменту разбора тела лямбды уже точно знаем тип x, и поэтому можем проверить корректность выражения.

Ещё одна тонкость, которая может вызвать недоумение. Рассмотрим объявления двух функций:

```
let twice x = (x + x)
let threeTimes x = (x + x + x)
```

И добавим ещё одно объявление:

```
let sixTimesInt64 (x:int64) = threeTimes x + threeTimes x
```

И если теперь посмотреть в F# Interactive, можно увидеть очень странные вещи:

```
val twice : x:int -> int
val threeTimes : x:int64 -> int64
val sixTimesInt64 : x:int64 -> int64
```

Во-первых, twice почему-то int, хотя казалось бы, почему? Ведь она могла бы с тем же успехом быть int64, float или вообще любого типа, поддерживающего сложение, а почему-то вывелась в int. Во-вторых, threeTimes, хотя ничем внешне не отличается, вывелась в int64. Только потому, что sixTimesInt64 вызывает её с 64-битным аргументом. А что будет, если написать такую же sixTimesFloat? Внезапно, ошибка компиляции.

Связано такое странное поведение с тем, что арифметические операции не генерики, и в уравнениях над типами нет возможности выразить, что тип должен поддерживать такой-то оператор. Оператор «+» просто перегружен для разных типов, и если используется в контексте, где тип неизвестен, то предполагается, что его аргументы int (просто потому, что они должны быть какого-то конкретного типа, а int — самый частоиспользуемый). Если же из контекста известен конкретный тип аргумента, используется перегрузка оператора, соответствующая этому типу. Поэтому и могут наблюдаться забавные эффекты:

```
let myNumericFn x = x * x
myNumericFn 10
myNumericFn 10.0  // Не скомпилится

let myNumericFn2 x = x * x
myNumericFn2 10.0
myNumericFn2 10  // Не скомпилится
```

Вообще, типизация по первому использованию иногда используется как дешёвая замена настоящего Хиндли-Милнера. Например, малоизвестный язык Haxe (https://haxe.org/), с которым, однако же, автору приходилось сталкиваться по долгу службы, использует именно такой подход к типизации вообще. Плюсы этого — простая реализация компилятора, простые и понятные программисту правила типизации. Минусы — дописали куда-нибудь вызов функции, вся программа перестала компилироваться с парой сотен ошибок типов. В F# такая типизация используется только для арифметических операций и только потому, что по-другому сделать очень сложно технически.

3. Встроенные шаблонные операции

Итак, арифметические операции не генерики, однако в стандартной библиотеке есть довольно много генерик-операций, иногда даже если это совершенно неочевидно, как реализовать. Например, операции сравнения:

```
val (=) : 'a -> 'a -> bool
val (<) : 'a -> 'a -> bool
val (<=) : 'a -> 'a -> bool
val (>) : 'a -> 'a -> bool
val (>=) : 'a -> 'a -> bool
```

Также генерики функции, относящиеся к сравнению

```
val compare : 'a -> 'a -> int
val (min) : 'a -> 'a -> 'a
val (max) : 'a -> 'a -> 'a
```

Казалось бы, ничего странного в этом нет, в C# достаточно реализовать IComparable и будет то же самое, но в F# оно автоматически работает для всех встроенных типов:

```
> ("abc", "def") < ("abc", "xyz");;
val it : bool = true
> compare (10, 30) (10, 20);;
val it : int = 1

> compare [10; 30] [10; 20];;
val it : int = 1
> compare [| 10; 30 |] [| 10; 20 |];;
val it : int = 1
> compare [| 10; 20 |] [| 10; 30 |];;
val it : int = -1
```

Это касается и любых комбинаций встроенных типов, так что списки пар или массивы списков четвёрок будут без каких-либо дополнительных усилий сравниваться операторами сравнения, для них будет даже считаться минимум и максимум. Сравнение лексикографическое, сравниваемые объекты обходятся рефлексией и сравниваются поэлементно. Это на самом деле довольно удобно — пока ваш код использует встроенные типы, большая

часть работы уже сделана в самом языке. Единственное, что можно прострелить себе ногу — например, в качестве ключа хеш-таблицы может быть очень плохой идеей использовать список произвольной длины. Сравнение для него будет работать, но требует просмотра всего списка — так что вам не надо будет прикладывать никаких усилий (вы этого можете даже не заметить), а работать будет очень медленно.

Для своих классов, однако, потребуется честно реализовать IComparable, внутрь классов F# не смотрит. Поэтому внутри F#-кода (который точно не будет вызываться из C#) предпочтительнее встроенные типы.

Так же работает и генерик-печать:

```
> sprintf "result = %A" ([1], [true]);;
val it : string = "result = ([1], [true])"
```

Форматный спецификатор %A не вызывает ToString(), а пытается рекурсивно обойти значение и построить человекочитаемое представление. Опять-таки, внутрь объектов пользовательских классов оно не идёт, вызывая ToString() вместо обхода. ToString() можно явно вызывать и вручную с помощью форматного спецификатора %O:

```
> sprintf "result = %O" ([1], [true]);;
val it : string = "result = ([1], [true])"
```

Для встроенных типов разницы на самом деле не будет, потому что для них ToString() определяется через структурный обход, как %A. Для пользовательских классов тоже, потому что %A для них просто вызывает ToString, как %O, так что различие возможно разве что в скорости работы.

Ещё есть встроенный генерик-boxing/unboxing. Напомним, что boxing — это «упаковка» объекта произвольного типа в ссылочный тип и размещение его на куче, unboxing — наоборот, распаковка упакованного объекта и размещение его на стеке вызовов. Если объект и так ссылочного типа (например, строка или массив, или любой пользовательский класс), boxing и unboxing ничего не делают. Если объект типа-значения (например, int), то boxing создаст object с одним полем типа int, а unboxing возьмёт этот object и вернёт int:

```
> box 1;;
val it : obj = 1

> box "abc";;
val it : obj = "abc"

> let sobj = box "abc";;
val sobj : obj = "abc"

> (unbox<string> sobj);;
val it : string = "abc"

> (unbox sobj : string);;
val it : string = "abc"
```

Эти операции нужны, когда кто-то требует значение ссылочного типа, а у нас есть тип-значение. Например, в Java любая генерик-коллекция может хранить только значения ссылочного типа (там используется стирание типов для симуляции генериков, так что по-другому там не сделать), поэтому список int-ов в Java — это на самом деле список Object-ов с int-ами внутри, и любая операция с таким списком включает в себя автоматический boxing или unboxing (что бесконечно огорчает сборщик мусора, которому потом чистить эти Object-ы). В .NET генерики могут хранить в себе типы-значения, поэтому boxing/unboxing требуется гораздо реже, но иногда всё-таки бывает нужен, например, API сериализации умеет работать только со ссылочными типами:

```
open System.IO
open System.Runtime.Serialization.Formatters.Binary

let writeValue outputStream (x: 'a) =
    let formatter = new BinaryFormatter()
    formatter.Serialize(outputStream, box x)

let readValue inputStream =
    let formatter = new BinaryFormatter()
    let res = formatter.Deserialize(inputStream)
    unbox res
```

При вызове formatter.Serialize требуется явный вызов box, чтобы гарантировать, что генерик-параметр x в Serialize передавался именно как ссылочный тип. А в readValue, чтобы получить свой тип обратно, надо в конце вызвать unbox (опять-таки, если 'a и так ссылочный тип, box и unbox просто ничего не сделают). box и unbox сами генерики, и автоматически работают для любого встроенного типа, а пользовательские типы по определению всегда ссылочные, так что box и unbox работают без каких-либо дополнительных усилий и для них.

Вот пример использования сериализации, написанной выше:

```
let addresses = Map.ofList [
    "Jeff", "123 Main Street, Redmond, WA 98052";
    "Fred", "987 Pine Road, Phila., PA 19116";
    "Mary", "PO Box 112233, Palo Alto, CA 94301" ]

use fsOut = new FileStream("Data.dat", FileMode.Create)
writeValue fsOut addresses
fsOut.Close()

use fsIn = new FileStream("Data.dat", FileMode.Open)
let res : Map<string, string> = readValue fsIn
fsIn.Close()
```

Тут у нас есть какая-то Map из трёх пар «ключ-значение» (кстати, обратите внимание, каждая строка списка — это отдельная пара, там внутри запятая). Мы её пишем в файл Data.dat, вызывая writeValue, а затем читаем, вызывая readValue.

С readValue есть тонкость, связанная с её типом: formatter.Deserialize не имеет идей, значение какого типа он десериализует. Вместе с тем readValue тоже генерик, возвращающий значение неизвестного типа 'a. Поэтому если бы мы написали `let res = readValue fsIn`, получили бы ошибку компиляции. Тут нужна явная аннотация типа возвращаемого значения, чтобы она позволила определить тип возвращаемого значения readValue a из этого параметр-тип для unbox. По этим же причинам мы явно передавали параметр-тип в unbox или писали аннотации типов в примере выше. Чем хороша система типов F# — она не требует аннотации типа в каком-то определённом месте, лишь бы компилятор мог как-то понять, каков должен быть тип возвращаемого значения у unbox.

4. Обобщение кода

То, что в F# есть автоматическое обобщение, сильно помогает, но если про генериковость не думать, код может всё равно получиться не очень генериком, что может быть плохо. Например, рассмотрим алгоритм Евклида нахождения наибольшего общего делителя (Highest Common Factor, если по-английски):

```
let rec hcf a b =
    if a = 0 then b
    elif a < b then hcf a (b - a)
    else hcf (a - b) b
```

По идее алгоритм Евклида должен работать в любом евклидовом кольце, но система типов говорит, что написанная нами функция применима только к int-ам:

```
val hcf : int -> int -> int
```

Её можно даже повызывать и проверить, что она работает, но только для int-ов:

```
> hcf 18 12;;
val it : int = 6

> hcf 33 24;;
val it : int = 3
```

Проблема, очевидно, в том, что 0 является int-овым литералом, операция «=» принимает первым и вторым аргументом обязательно один и тот же тип, и таким образом a тоже int. А раз a сравнивается с b, то и b int (тоже, сравнение имеет тип 'a -> 'a -> 'a, так что типы аргументов должны быть одинаковы.

Побороться с этим можно, параметризовав функцию, так, чтобы значение, считающееся в данном кольце нулём, передавалось явно. А поскольку мы можем захотеть применять алгоритм Евклида, например, к кольцу многочленов, надо передавать и операции:

```
let hcfGeneric (zero, sub, lessThan) =
    let rec hcf a b =
        if a = zero then b
        elif lessThan a b then hcf a (sub b a)
        else hcf (sub a b) b
    hcf
```

Теперь тип у этой функции правильно выведется в генерик:

```
val hcfGeneric: 'a * ('a -> 'a -> 'a) * ('a -> 'a -> bool)
    -> ('a -> 'a -> 'a)
```

То есть это функция, принимающая значение (нейтральный элемент кольца), функцию вычитания и функцию «меньше», и возвращающую функцию, реализующую алгоритм Евклида. Так что на самом деле мы сделали не функцию, а фабрику функций, которую можно использовать, чтобы получить алгоритм Евклида для конкретного кольца:

```
let hcfInt = hcfGeneric (0, (-), (<))
let hcfInt64 = hcfGeneric (0L, (-), (<))
let hcfBigInt = hcfGeneric (0I, (-), (<))
```

Кстати, обратите внимание, что кажется, будто мы $-$ и $<$ передаём одинаковыми. На самом деле, это не так: $-$ вообще не генерик, поэтому конкретная перегрузка определяется из контекста, который тут получается довольно хитро. В hcfInt, например, мы передаём int-овый 0, тип a из сравнения с ним определяется как int, a (sub b a) заставляет sub быть int - int, и *поэтому* для $-$ выбирается int-овая перегрузка. Так что тут, чтобы определить тип параметра, мы должны параметр с известным типом «протянуть» внутрь функции, и потом уже вернуться к месту вызова со знанием ограничений на второй параметр, Хиндли-Милнер во всей красе.

А вот $<$ генерик, поэтому и вправду одинаковый для всех трёх функций.

Такой приём, с приёмом в качестве параметров операций для того, чтобы сделать алгоритм возможно более генериком, часто применяется в функциональном программировании, и даже может быть уточнён до, не побоюсь этого слова, паттерна «Словарь операций»:

```
type Numeric<'a> =
    { Zero: 'a;
      Subtract: ('a -> 'a -> 'a);
      LessThan: ('a -> 'a -> bool); }

let hcfGeneric (ops : Numeric<'a>) =
    let rec hcf a b =
        if a = ops.Zero then b
        elif ops.LessThan a b then hcf a
            (ops.Subtract b a)
        else hcf (ops.Subtract a b) b
    hcf
```

То есть вместо передачи просто какого-то невнятного кортежа мы создаём запись, где у каждого требуемого элемента есть адекватное имя и в явном виде выписанный тип, что дольше писать, но, скорее всего, повысит сопровождаемость программы. Вот тип того, что получилось:

```
val hcfGeneric : Numeric<'a> -> ('a -> 'a -> 'a)
```

То есть это всё так же фабрика функций, но принимающая значение типа Numeric, которое само генерик с одним параметром-типом. Вот как это дело можно использовать:

```
let intOps = { Zero = 0;
    Subtract = (-);
    LessThan = (<) }

let bigintOps = { Zero = 0I;
    Subtract = (-);
    LessThan = (<) }

let hcfInt = hcfGeneric intOps
let hcfBigInt = hcfGeneric bigintOps
```

И вот что получится:

```
val hcfInt : (int -> int -> int)
val hcfBigInt : (bigint -> bigint -> bigint)

> hcfInt 18 12;;
val it : int = 6

> hcfBigInt 181028711616223238303957 6I
    12390281782930928304802390321;;
val it : bigint = 33224I
```

На самом деле, можно провести некоторые аналогии между словарями операций и таблицей виртуальных методов. В объектно-ориентированном программировании есть паттерн «Шаблонный метод», который говорит, что можно описать алгоритм в базовом классе в терминах виртуальных методов, и в классах-потомках переопределить эти виртуальные методы так, чтобы общий алгоритм делал что-то конкретное (например, шаблонный алгоритм может заниматься инициализацией системы, а перегружаемые виртуальные методы могут соответствовать разным конфигурациям этой самой системы). Ну вот, алгоритм Евклида — шаблонный метод, вместо наследования — словарь операций.

Вообще, словарь операций может таскать с собой набор функций, который можно подсунуть типу (не важно, функции, классу, чему угодно) и тем изменить его поведение, при этом не изменяя сам тип. Сравните это с определением полиморфизма. Плюс словари операций можно менять во время выполнения, просто передавая объекту другой словарь, чего традиционное наследование никак не может. В общем, «виртуальные методы» в функциональных языках вполне можно хранить отдельно от данных, и это позволяет реализовать полиморфизм, только гибче. В F# есть самые настоящие классы, наследование и полиморфизм, но используются они в чисто F#-овом коде гораздо реже, чем в C#, как раз потому, что есть более интересные альтернативы типа словарей операций (ну или просто возможности легко и просто передать функцию).

5. Наследование и генерики

5.1. Приведение типов

Поскольку F# — ещё и объектно-ориентированный язык программирования, возникают некоторые тонкости, связанные со взаимодействием генериков и обычного ООПшного наследования. Но для начала, операторы повышающего и понижающего каста. Повышающий каст в F# пишется вот так:

```
> let xobj = (1 :> obj);;
val xobj : obj = 1

> let sobj = ("abc" :> obj);;
val sobj : obj = "abc"
```

Каст типа-значения к ссылочному типу (в данном примере obj, но если вы помните иерархию типов .NET, то тип-значение в принципе может быть откастан из ссылочных типов только к obj) автоматически выполняет boxing. Каст ссылочного к ссылочному, естественно, не выполняет, поэтому sobj из второго примера указывает на тот самый объект в памяти, где лежит строка abc. В первом примере xobj — это новый объект на куче.

Понижающий каст:

```
> let boxedObject = box "abc";;
val boxedObject : obj

> let downcastString = (boxedObject :?> string);;
val downcastString : string = "abc"

> let xobj = box 1;;
val xobj : obj = 1

> let x = (xobj :?> string);;
error: InvalidCastException raised at or near stdin:(2,0)
```

Его принципиальное отличие от повышающего каста в том, что он может быть неудачным, и во время компиляции это проверить невозможно (поэтому и вопросик в записи оператора). Если тип времени выполнения объекта не подходящий, то во время выполнения бросится InvalidCastException, как в третьем примере. То есть понижающий каст ведёт себя как «C++-style cast» в C#. Аналогом оператора as из C# является match с шаблоном проверки типа:

```
let checkObject (x: obj) =
    match x with
    | :? string -> printfn "The object is a string"
    | :? int -> printfn "The object is an integer"
    | _ -> printfn "The input is something else"
```

Однако мы, наверное, хотим не просто проверить тип, а проверить и сделать что-нибудь с откастанным значением, так что чаще всего шаблоны проверки типа используются с as (F#-овским, объявление имени для поматченного шаблона):

```
let reportObject (x: obj) =
    match x with
    | :? string as s ->
        printfn "The input is the string '%s'" s
    | :? int as d ->
        printfn "The input is the integer '%d'" d
    | _ -> printfn "the input is something else"
```

5.2. Гибкие ограничения

Однако вернёмся к генерикам. Как и в C#, генерики в F# могут накладывать ограничения на параметры-типы, чтобы потом иметь возможность что-нибудь с ними делать (к слову, ограничения точно такие же, как в C#, потому что это функциональность больше байткода .NET, чем языка). Например:

```
> open System.Windows.Forms;;
> let setTextOfControl (c : 'a when 'a :> Control)
    (s:string) = c.Text <- s;;
val setTextOfControl: #Control -> string -> unit
```

Эта запись означает, что setTextOfControl генерик, параметр-тип которого может быть любым типом, наследующимся от Control. А раз мы знаем, что он наследуется от Control, то у него есть свойство Text, куда мы можем выполнить присвоение. Интерпретатор выдал тип этой функции с #Control, неспроста — это сокращённая форма записи ограничения «является наследником», так что пример выше можно было бы записать вот так:

```
> open System.Windows.Forms;;
> let setTextOfControl (c : #Control) (s:string) =
    c.Text <- s;;
val setTextOfControl: #Control -> string -> unit
```

В данном случае это не очень полезно, потому что при вызове функции компилятор выполняет повышающий каст автоматически, но когда тип, который мы хотим сделать «гибким», является частью сложного типа, без # не обойтись:

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)
```

```
// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

© https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types

Это активно используется и в библиотечных функциях:

```
val concat: sequences:seq<#seq<'T>> -> seq<'T>
```

В функцию с такой аннотацией типа можно передавать любых наследников seq, то есть практически любые коллекции:

```
let list1 = [1;2;3]
let list2 = [4;5;6]
let list3 = [7;8;9]

let concat1 = Seq.concat [ list1; list2; list3]
printfn "%A" concat1

let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = [|7;8;9|]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3

let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

© https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types

Это самое близкое, что есть к вариантности в F# (ключевым словам in и out в C#). Достаточно хорошо для большого количества случаев, а поскольку в функциональном программировании наследование вообще не любят, поддержкой настоящей ковариантности/контравариантности никто не заморачивался (хотя такой issue в репозитории компилятора открыт).

6. Отладка типов

Автоматический вывод типов, автоматическое обобщение и прочие хорошие вещи имеют один серьёзный недостаток: нетривиальную диагностику ошибок типов. Есть даже несколько отдельных приёмов, позволяющих быстро понять, что не нравится компилятору и быстро это поправить. Справедливости ради, диагностика в современном F# существенно лучше, чем в ранних версиях компилятора. Бывали времена, когда при ошибке типизации подсвечивалась функция (или даже программа) целиком и выводилось сообщение в духе «тут что-то не так с типами». Сейчас такого нет, но всё равно есть ошибки и соответствующие им диагностические сообщения, которые могут быть попросту непонятны непосвящённому.

6.1. Неопределённый тип

Начнём, однако, с простого. Самая частая ошибка типизации у людей, привыкших к объектно-ориентированному программированию — недостаточная информация о типе при вызове метода или обращении к полю. Например:

```
> let transformData inp =
    inp |> Seq.map (fun (x, y) -> (x, y.Length));;

inp |> Seq.map (fun (x, y) -> (x, y.Length))
----------------------------------------^^^^^^^^
stdin(11,36): error: Lookup on object of indeterminate
type. A type annotation may be needed prior to this
program point to constrain the type of the object.
This may allow the lookup to be resolved.
```

Здесь действительно неочевидно, с чего мы взяли, что у имеет свойство Length (или это вообще метод и мы хотим получить лямбду как второй элемент пары?). Однако наибольшую конфузию это вызывает, когда тип вроде бы понятен из контекста:

```
Seq.map (fun x -> x.Length) ["a"; "b"]
```

Это мы уже обсуждали, когда речь шла про алгоритм вывода типов — напомню, что F# не умеет фиксировать ограничение на переменную типа вида «имеет такой-то метод/свойство/поле», поэтому компилятор сначала ругается, что вызов не может быть проверен, а уже потом узнаёт, что на самом деле всё ок и x — это строка. Надо с этим жить, благо это легко исправляется аннотациями типов:

```
let transformData inp =
    inp |> Seq.map (fun (x, y: string) -> (x, y.Length))
```

6.2. Уменьшение общности

Следующая ошибка более интересная:

```
let printSecondElements (inp : seq<'a * int>) =
    inp
    |> Seq.iter (fun (x, y) -> printfn "y = %d" x)
```

Тут даже есть явная аннотация типа, и вроде всё хорошо, однако при компиляции получаем ошибку:

```
|> Seq.iter (fun (x, y) -> printfn "y = %d" x)
----------------------------------------^^^^^^^^^
stdin(21,38): warning: FS0064: This construct causes
code to be less generic than indicated by the type
annotations. The type variable 'a has been
constrained to the type 'int'.
```

Ошибка указывает на место, которое вызывает «срезку типа» генерика до конкретного типа — мы заказывали seq<'a * int>, а получили seq<int, int>. Внимательный читатель, наверное, уже понял, где ошибка, но в реальной жизни такая «срезка» может быть гораздо менее очевидной, так что это можно отладить хорошим приёмом. Раз наш тип должен быть генериком, то если мы вместо параметра-типа 'a подставим какой-нибудь конкретный тип, он всё равно должен будет работать:

```
type PingPong = Ping | Pong

let printSecondElements (inp : #seq<PingPong * int>) =
    inp |> Seq.iter (fun (x, y) -> printfn "y = %d" x)
```

Желательно именно какой-нибудь свой тип, чтобы он точно не оказался случайно с чем-то совместим. Поскольку у нас в коде баг, мы получим ошибку компиляции, но гораздо более внятную:

```
|> Seq.iter (fun (x,y) -> printfn "y = %d" x)
-------------------------------------------------^
stdin(27,47): error: FS0001: The type 'PingPong' is not
compatible with any of the types byte, int16, int32,
int64, sbyte, uint16, uint32, uint64, nativeint,
unativeint, arising from the use of a printf-style
format string
```

Тут уже совершенно понятно, что мы опечатались в имени переменной и написали x вместо y.

6.3. Value Restriction

И наконец, самая загадочная ошибка:

```
> let empties = Array.create 100 [];;
------^^^^^^^^
error: FS0030: Value restriction. Type inference
```

```
has inferred the signature
val empties : '_a list []
but its definition is not a simple data constant.
Either define 'empties' as a simple data expression,
make it a function, or add a type constraint
to instantiate the type parameters.
```

Вот тут у вчерашнего C# программиста может возникнуть полное непонимание происходящего. Во-первых, что говорит компилятор: '_a — это обозначение *недообобщённой* переменной типа. Компилятор полагает, что '_a должна была в этом месте разрешиться в конкретный тип, но это параметр-тип генерика, который в данном контексте неуместен, о чём нам и сообщают.

Во-вторых, почему ему это не нравится: Array — это мутабельный тип, [] означает «пустой список неизвестного типа». Мы создали массив из сотни пустых списков — окей, давайте теперь положим туда в нулевую ячейку список [1]. А потом в первую ячейку список ["a"]. Это всё должны быть корректные операции, потому что во время компиляции мы не имеем идей, что окажется в массиве. Но тогда какого типа массив, и что будет, если написать `let x: int = empties[1]`? Получается, что во время компиляции тип массива мы посчитать не можем, и это жыра в системе типизации, которую не может себе позволить функциональный язык. Поэтому такие объявления попросту запрещены и мутабельные значения никогда не могут быть генериками.

Однако вот такие определения вполне валидны и компилируются:

```
let emptyList = []
let initialLists = ([], [2])
let listOfEmptyLists = [[]; []]
let makeArray () = Array.create 100 []

val emptyList : 'a list
val initialLists : ('a list * int list)
val listOfEmptyLists : 'a list list
val makeArray : unit -> 'a list []
```

- emptyList — значение генерик-типа, но не мутабельного. Да, мы не знаем его точный тип, но мы не можем туда ничего присвоить, поэтому точно знаем, что это всегда пустой список, и все операции с ним можно проверить статически.
- initialLists — то же самое, более сложная структура, тоже генерик, но не мутабельна.
- listOfEmptyLists — ближе всего к исходному примеру, но, как нетрудно догадаться, тоже немутабельная структура данных, поэтому типизацию не ломает. Всё портится, когда мы можем положить в контейнер значение одного типа, а достать другого — поэтому если мы не можем ничего положить, то и проблемы нет.
- makeArray — это функция, которая буквально возвращает исходный пример, но почему-то компилится. Почему? Дело в том, что если мы напишем `let empties = makeArray ()`, получим value restriction уже на empties, а `let empties: int list [] = makeArray ()` вообще прекрасно скомпилируется. То есть

value restriction — это не проблема функции, функция немутабельна и может быть генериком. А вот значение, которое она возвращает, мутабельно и не может.

Как с value restriction бороться: во-первых, как уже показывали, добавить негенериковую аннотаци типа (то есть вручную уменьшить общность):

```
let empties : int list [] = Array.create 100 []
```

Во-вторых, сделать из значения функцию. Например,

```
let mapFirst = List.map fst
```

— казалось бы mapFirst — функция, она немутабельна и может быть генериком. Но нет, для компилятора это значение функционального типа, тип выведется в `('_a * '_b) list -> '_a list`, соответственно, компилятор будет расстроен его недообобщённостью. Почему — компилятор не может доказать, что справа от присваивания выражение, не имеющее побочных эффектов. Починить это можно, сделав mapFirst явной функцией:

```
let mapFirst inp = List.map fst inp
```

(η-преобразование в обратную сторону, если вы ещё помните лямбда-исчисление). Теперь это функция, функция может быть генериком, всё хорошо.

Более хитрый пример в том же ключе:

```
let printFstElements =
    List.map fst
    >> List.iter (printf "res = %d")
```

Тут мы знаем про тип функции, что она принимает список пар (раз fst), и первый элемент пары — int (раз %d). А вот про второй элемент пары ничего не знаем:

```
((int * '_a) list -> unit)
```

Соответственно, получаем value restriction, как и в предыдущем случае. Лечится так же:

```
let printFstElements inp = inp
    |> List.map fst
    |> List.iter (printf "res = %d")
```

Теперь функциональное значение с точки зрени компилятора является обычной функцией и для неё уже параметр-тип разрешён.

Ну и последний способ может показаться уж совсем неожиданным:

```
let emptyLists = Seq.init 100 (fun _ -> [])
```

— так не скомпилируется по уже обсуждавшимся здесь причинам, про чистоту Seq.init ничего не известно.

А так:

```
let emptyLists<'a> : seq<'a list> = Seq.init 100 (fun _ -> [])
```

— скомпилируется, что? Ничего же не поменялось. А дело в том, что listId скомпилируется теперь не в значение, а в генерик-функцию от параметра-типа, что позволит иметь столько разных listId, сколько раз мы инстанцировали этот генерик. Это может быть совершенно непонятно, но так устроен компилятор, и вот тут: https://habr.com/ru/company/microsoft/blog/348460/ хорошее объяснение, как именно это работает (вообще, там хороший рассказ про особенности value restriction, рекомендую прочитать).

И теперь мы можем вручную инстанцировать нужные генерики, избежав тем самым ошибок типизации:

```
> Seq.length emptyLists;;
// Seq.length не интересны значения внутри, поэтому и довывести тип не нужно
val it : int = 100

> emptyLists<int>;;
val it : seq<int list> = seq [[]; []; []; []; ...]

> emptyLists<string>;;
val it : seq<string list> = seq [[]; []; []; []; ...]
```

6.4. Point-free

С Value Restriction напрямую связан стиль программирования, пришедший из чисто функциональных языков типа Haskell — Point-free-программирование. Его идея в том, что раз уж у нас функциональный язык, можно думать и писать программу в терминах не преобразования данных, как все, возможно, привыкли, а в терминах копозиции функций, как, возможно, привыкли любители функционального анализа. Один пример программы в таком стиле мы уже видели:

```
let printFstElements =
    List.map fst
    >> List.iter (printf "res = %d")
```

Чтобы распечатать все первые элементы пар из списка пар, надо взять List.map от fst и склеить её с функцией List.iter, которая распечатает значения, применив к каждому элементу printf, частично параметризованный форматной строкой. Обратите внимание, у printFstElements нет параметра, и более того, список, который мы печатаем, вообще нигде не упоминается. Собственно, стиль называется pint-free, потому что в программе не упоминаются точки, в которых должно быть посчитано значение функции: $f(x)$ в матане может означать значение функции f в точке x, так что $f(x)$ — это «point-full»-нотация, а f — point-free.

Вот как можно обычную программу переделать под point-free-стиль. Положим, у нас зачем-то есть функция, которая оставляет в списке только те элементы, у которых первый элемент пары больше нуля:

```
let fstGt0 xs = List.filter (fun (a, b) -> a > 0) xs
```

Первое, что мы можем сделать — это применить η-преобразование и избавиться от xs:

```
let fstGt0'1 : (int * int) list -> (int * int) list =
    List.filter (fun (a, b) -> a > 0)
```

Тут нас уже поразит value restriction, поэтому придётся явно выписывать аннотации типов. Одним аргументом стало меньше. Однако внутри лямбды два аргумента, и избавиться от них не так просто. Для начала сделаем так, чтобы аргумент был один, «схлопнув» пару в одно значение:

```
let fstGt0'2 : (int * int) list -> (int * int) list =
    List.filter (fun x -> fst x > 0)
```

Стало меньше точек, но от x уже так просто не избавиться, потому что он используется где-то внутри выражения, так что η-преобразование не применить. Постараемся вытащить его в крайнюю праву позицию, переписав тело лямбды:

```
let fstGt0'3 : (int * int) list -> (int * int) list =
    List.filter (fun x -> ((<) 0 << fst) x)
```

`fst x > 0` мы переписали сначала как `0 < fst x`, затем записали оператор $<$ в префиксной форме: `(<) 0 (fst x)`, затем вспомнили про каррирование: `((<) 0) (fst x)`, затем применили композицию (тут фактически записано $f(g(x))$, где f — это `((<) 0)`, а g — fst), получив `((<) 0 << fst) x`. Ну а дальше дело техники:

```
let fstGt0'4 : (int * int) list -> (int * int) list =
    List.filter ((<) 0 << fst)
```

Всё это хорошо, но должен был возникнуть вопрос *«зачем?»*. А, в общем-то, незачем. Программы в point-free-стиле имеют тенденцию быть существенно короче, однако существенно тяжелее для восприятия и сложнее в отладке. Поэтому в продакшн-коде за чрезмерную любовь к point-free могут и уволить. Кстати, в Haskell, поскольку он чисто функциональный, нет понятия «value restriction» (ну, там вообще всё немутабельно), поэтому там point-free не требует дополнительной работы, поэтому более распространён (если интересно, погуглите, например, «комбинатор «Сова»). F# в этом плане в каком-то смысле даже лучше Haskell, поскольку наказывает value restriction-ом сомнительную для индустриального программирования практику. Однако же point-free в малых дозах очень полезен, например, `fun x -> x` надо, конечно, писать в point-free-стиле как id. И вообще, везде, где можно не писать аргумент без ущерба понимабельности программы, лучше его не писать (опять же, не `fun (a, b) -> b`, а snd), но не увлекаться.

Point-free, однако же, открывает глаза на систему типов, на взаимосвязь пайпа и композиции, вообще на программу как комбинацию функций, а не конвейер по переработке данных, поэтому тут про него так подробно — это очень полезно в образовательных целях, так что хоть немного point-free-программирования каждый должен попробовать.