Teaching Functional Programming to Professional .NET Developers

Tomas Petricek[1]

[1] University of Cambridge, Cambridge, United Kingdom
tp322@cam.ac.uk

Abstract. Functional programming is often taught at universities to first-year or second-year students and most of the teaching materials have been written for this audience. With the recent rise of functional programming in the industry, it becomes important to teach functional concepts to professional developers with deep knowledge of other paradigms, most importantly object-oriented.

We present our experience with teaching functional programming and F# to experienced .NET developers through a book Real-World Functional Programming and commercially offered F# trainings. The most important novelty in our approach is the use of C# for relating functional F# with object-oriented C# and for introducing some of the functional concepts.

By presenting principles such as immutability, higher-order functions and functional types from a different perspective, we are able to build on existing knowledge of professional developers. This contrasts with a common approach that asks students to forget everything they know about programming and think completely differently. We believe that our observations are relevant for trainings designed for practitioners, but perhaps also for students who explore functional relatively late in the curriculum.

1 Introduction

Until recently, object-oriented languages such as Java, C# and C++ were dominating the industry. Despite this fact, functional languages have been successfully taught at universities and interesting approaches to teaching functional languages have been developed. Professionals approached functional programming with the intention to learn about different style of thinking, with the aim of becoming better developers by broadening their horizons.

With the recent rise of mixed functional languages such as F# and Scala and the inclusion of functional features in main-stream languages, the audience interested in functional programming changes. Developers more often want to add pragmatic functional programming to their toolbox, learn how it can be applied to their daily tasks and understand how it relates to the patterns they regularly use.

We believe that the most efficient way to teach functional programming to this new audience is to relate functional programming to what they already know and build on this existing knowledge. In particular, we discuss the following approaches:

Explaining concepts. Functional programming concepts are often explained using parallels with mathematics. We show that compositionality, immutability and first-class functions can be related to common programming patterns, such as expression builder and problems encountered in object-oriented languages (Section 2).

Understanding style. At a larger scale, we show how to explain concepts behind functional application design. We discuss how functional programming relates to domain modeling (Section 3.1) and relations between functional style and design patterns.

Functional programming in practice. Despite the interest in functional style, many companies still use mostly object-oriented languages like C#. We demonstrate several techniques that developers can learn through functional programming, but that can be efficiently used in other languages (Section 4).

2 Learning functional concepts via C#

Functional programming is based on different core principles than familiar programming styles. When introducing functional languages to first-year or second-year students, this is not a problem, because they can easily accept different set of principles. However, this causes notable burden for professionals who are trained in (and comfortable with) thinking using imperative and object-oriented terms. Therefore we should avoid saying "forget everything you learned before" and instead find a way to explain functional concepts in terms of familiar imperative and object-oriented terms. In this section, we explain analogies that we use for core concepts such as immutable types and higher-order functions.

2.1 Expression builder and immutability

In the object-oriented style, objects are constructed by invoking a constructor and then configuring the object by calling methods (better modularity is possible using Dependency Injection [1], but we consider a simple case). The object construction syntax tends to be cumbersome, which can be avoided using method chaining:

```
var tea = Product.Create("Earl Gray Tea")
                 .WithPrice(10.0M).WithPromotion();
```

The above syntax is also called Fluent Interface pattern [2]. It is obtained by defining methods, such as `WithPrice`, that set a property of the object and then return the same instance. As a result, another method can be called immediately on the result. The construction can be wrapped in a separate Builder object [3]. Such combination has been called Expression Builder pattern by Fowler [4].

For our explanation, it is sufficient to consider `Product` that contains builder methods directly. In standard object-oriented version, the class contains private mutable fields (`name`, `price`, etc.). Methods such as `WithPrice` are implemented as follows:

```
public Product WithPrice(string price) {
  this.price = price;
  return this;
}
```

In object-oriented practice, this is a common (and useful) pattern. However, it reveals a problem with using mutable state that is not always obvious to the users of the pattern. Consider the following example that creates two products:

```
var p1 = Product.Create("Earl Gray Tea").WithPrice(10.0M);
var p2 = Product.Create("Earl Gray Tea").WithPrice(12.0M);
```

We might refactor the code according to the “Do not Repeat Yourself” principle [5]:

```
var p  = Product.Create("Earl Gray Tea");
var p1 = p.WithPrice(10.0M);
var p2 = p.WithPrice(12.0M);
```

However, the refactored version behaves differently! It creates only a single product and the two `WithPrice` calls mutate the same instance. Indeed, this flaw can be fixed by using immutable types:

```
public Product WithPrice(decimal price) {
  return new Product(this.name, price);
}
```

In C#, we mark fields as `readonly`, to prevent assignments outside of the constructor. The `WithPrice` method now returns a new instance using a (private) constructor. After changing the object to immutable the above refactoring preserves the semantics.

Example summary. The development discussed so far illustrates the importance of *compositionality*, which is a key concept of functional programming. We also demonstrated that compositionality can be obtained by using *immutability*, which is a typical approach in functional languages. Aside from compositionality, it is worth noting that processing of immutable objects can be safely parallelized. If the two `WithPrice` calls were expensive, they could be run concurrently.

In Java-like languages, the immutable Expression Builder pattern has to be implemented by hand, but an equivalent F# code can be written easily using record types:

```
type Product = { Name : string; Price : decimal }
```

An example that constructs two products without duplicating code looks as follows:

```
let p = { Name = "Earl Gray Tea"; Price = 0.0M }
let p1 = { p with Price = 10.0M }
let p2 = { p with Price = 12.0M }
```

This comparison relates an advanced, but useful object-oriented pattern to a basic F# type. Anecdotally, this demonstrates that the defaults in functional languages make it easier to write correct code. In our experience, starting with an easy-to-make mistake in object-oriented code makes the argument convincing to professional developers.

2.2 Hole in the middle and functions

Lambda functions are becoming more common in main-stream languages, including C#, Python and C++. As a result, many professional programmers are already familiar with the concept and appreciate its importance. The need for function abstraction arises in many real-world scenarios. The material for our course [6] is based on the "Hole in The Middle" pattern [7], which describes a simple compelling scenario.

Consider the following example that uses `ParsingService` object to extract the title of a web page and logs exceptions that may have occurred:

```
1: var svc = new ParsingService("http://tomasp.net");
2: try {
3:   Console.WriteLine(svc.GetTitle());
4: } catch (WebException e) {
5:   Logger.Report("Parsing tomasp.net", svc, e);
6: }
```

The same exception handling and logging code would be repeated for every use of the `ParsingService` type. Refactoring the repeating code into methods is difficult, because the scope of exception handler overlaps with repeating blocks (lines 1,2 and 4,5,6).

Finding a pure object-oriented solution to the problem proves to be difficult. We might use the Template Method pattern [3], but that requires implementing a new derived class for every user of our `ParsingService` type. Alternative solutions based on aspect-oriented programming [9] are more elegant, but are not main-stream.

When introducing functional concepts using C#, we can start from anonymous delegates (introduced in 2005) and continue to more succinct lambda expressions that are available in C# 3.0:

```
WithParsingService("http://tomasp.net", svc =>
  Console.WriteLine(svc.GetTitle()) );
```

Here, `WithParsingService` is a method taking `Action<ParsingService>` as an argument. This delegate represents a function taking `ParsingService` and returning `void`.

Example summary. The above example follows from a real-world problem that many C# or Java developers are familiar with. To those unfamiliar with lambda abstraction, it shows practical benefits of functional concepts. For others, it shows that they are already familiar with another core functional technique.

After showing the C# example, it is easy to rewrite the code to F#. We leverage the fact that F# supports imperative constructs, so the implementation code (using exception handling and side-effects) closely corresponds to C#. The use is also similar:

```
withParsingService (fun svc -> printfn "%s" (svc.GetTitle()))
```

The example is not purely functional, which makes it perhaps more accessible as an initial step from imperative and object-oriented languages. Starting with more standard functional examples (such as list processing using `map` and `filter`) introduces larger gap, so we find them more appropriate as the second step.

2.3 Values and extension methods

In object-oriented programming, all methods should be defined as members of some class. In the real life, many operations are defined as static utility methods, often because the class where they belong cannot be extended or because they do not logically belong to any class. C# 3.0 [10] solves the first case with *extension methods* and Java 8 comes with a similar proposal [11].

In C# 3.0, the feature is mainly used for adding collection processing methods to any collection implementing the `IEnumerable<T>` interface. By following similar coding style, we can easily introduce standard functional values in a way that looks familiar to C# developers. In Chapters 5 and 6 of Real-World Functional Programming [12], we implement a simple `Option<T>` type with a set of extension methods well-known to functional programmers. The simplest representation of the option type in C# is a class with the following public interface:

```
public class Option<T> {
  public Option();
  public Option(T some);

  public bool HasValue { get; }
  public T Value { get; }
}
```

In functional languages, the type can be implemented using a discriminated union, which prevents accessing `Value` property when it is not defined. An alternative, safer, encoding using class hierarchy is discussed in Section 3.2. Here, we use simple encoding and focus on providing operations for working with `Option<T>`.

Without changing the type itself (which may be in a compiled library), we want to add operations that simplify working with options. Inspired by collection processing operations from .NET, we may first add `ForEach` extension method that performs a specified action if the value is available:

```
public void ForEach(this Option<T> option, Action<T> action) {
  if (option.HasValue) action(option.Value);
}
```

Other operations known from functional languages can be implemented in a similar style, which makes for a good exercise. To demonstrate the usefulness of these operations, we define `Map` and `Bind` methods (or `Select` and `SelectMany` using the C# naming convention). The following code uses the methods to work with a fictional database:

```
DB.TryFindProduct(id).Bind(prod =>
    DB.TryFindCategory(prod.CategoryID).Map(cat =>
      prod.Name + ", " + cat.Name)).
  ForEach(info => Console.WriteLine(info));
```

The snippet uses methods `TryFindProduct` and `TryFindCategory` that return values of type `Option<T>` as results. After retrieving a product, we use `Bind` to obtain its category and `Map` to turn the pair into a formatted string.

Example summary. The C# version of the code is not idiomatic (and would be rarely used in practice), but it still serves two purposes. Firstly, it demystifies functional programming by showing how F# works under the cover. Secondly, it demonstrates how option types avoid infamous `NullReferenceException`. Assuming `TryFindProduct` returns `Product`, which is `null` if the search fails, one could easily write:

```
Product prod = DB.TryFindProduct(id);
Category cat = DB.TryFindCategory(prod.CategoryID);
Console.WriteLine(prod.Name + ", " + cat.Name);
```

Incorrect handling of `null` values is ubiquitous. Greater safety available thanks to option values is often a good motivation for interest in functional programming. The example can be also used to discuss monads and LINQ [18], but this is an advanced topic and, as discussed in Section 4.1, we return to it later in our material.

This example also moves from techniques that professional C# developers might use without knowing functional programming to an example that, arguably, would be only written by someone familiar with functional languages. We use it to shift attention to F# (explaining language features like pattern matching as needed). In F#, the above example would be written as follows:

```
DB.tryFindProduct(id)
|> Option.bind (fun prod ->
     DB.tryFindCategory(prod.CategoryID)
     |> Option.map(fun cat -> prod.Name + ", " + cat.Name))
|> Option.iter (printf "%s")
```

The example demonstrates several interesting aspects of F#, including the pipelining operator (`|>`) and partial function application (`printf "%s"`). We explain these as needed, but postpone the details until later in our functional programming course.

3 Relating functional and object-oriented design

The previous section discussed how to explain core concepts of functional languages using analogies between object-oriented C# and functional F#. We started with C# versions of functional code, which means that the students can learn new ideas without leaving a language they are comfortable with.

In this section, we discuss the next step – we move from language features to application design and we also move to samples in F# first and showing C# aside only to explain how the same design could be represented in the object-oriented style.

3.1 Functional types and domain modeling

When designing object-oriented systems, software developers often start by drawing the class structure. Similarly, functional programmers start by defining the data types of the system. This analogy provides a good way to introduce standard functional data types such as records and discriminated unions (algebraic data types).

Many teaching materials on functional programming introduce data types using computer science examples such as lists, trees or expressions. These provide simple examples familiar to computer scientists. For professional software developers, we prefer examples that model real-world systems from domains such as point of sale or finance, which show how to use functional programming in daily job.

For example, consider an application for a checkout counter. The application stores a list of checkout line commands (`LineItem`) that can be either scanned item, cancellation of an earlier item or tendering for various kinds of payments. F# type declarations that model the domain look as follows:

```
type Price = decimal
type Quantity = int
type Product = { Code:string; Name:string; Price:Price }

type Tender =
  | CashTender
  | CardTender of string
type LineItem =
  | SaleItem of int * Product * Quantity
  | TenderItem of Tender * Price
  | CancelItem of int
type Sale = list<ListItem>
```

The declarations use type aliases (`Price` and `Quantity`) to model primitive types that may change during the development. `Product` is represented as an F# record (introduced when talking about immutability, see Section 2.1). The most interesting types are `Tender` and `LineItem`, which introduce the discriminated union type. Finally, `Sale` is a list of items representing the entire transaction.

To draw an analogy with object-oriented solution, we now consider a class hierarchy that can be used to represent the same domain in C#. We start by visualizing the model using an UML class diagram shown in Figure 1.

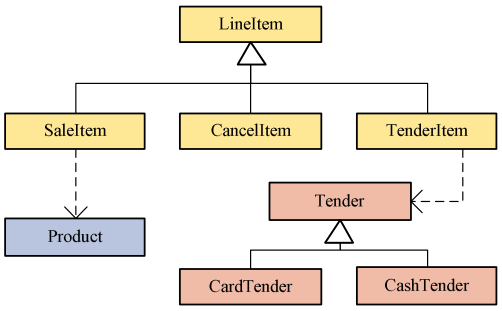

Figure 1. UML class diagram representing the point of sale domain.

The `Product` type, which was modeled using a record, is displayed as a class, showing that records are just a simple kind of classes – with just immutable properties. More interestingly, the two discriminated unions (`LineItem` and `Tender`) are modeled as class hierarchies. The `LineItem` type is represented as an (abstract) base class with subtype for every discriminated union case (`SaleItem`, `CancelItem` and `TenderItem`).

Example summary. We find that relating discriminated unions (algebraic data types) to class hierarchies in object-oriented programming is a useful explanation of the concept. For functional programmers, discriminated unions are simpler concept than class hierarchies, but existing C# developers are used to the other perspective. Moreover, F# discriminated unions are actually compiled as .NET class hierarchies, so showing the analogy suggests how the language works and how it can interoperate with C#.

Showing the class diagram side-by-side with the actual F# source code shows another important aspect of functional programming that is extremely valuable in practice. The domain model written in functional language fits on a "single page" and can be easily understood, often even by readers who do not know F#.

Unlike the UML diagram, functional type declarations are executable and are a part of the code base, so there is no danger that the model and the actual implementation are out of sync. The same cannot be done in Java or C#, because the diagram corresponds to 8 classes that would contain other functionality and would be stored in 8 separate files. The ability to model problem domains easily within the programming language is a benefit that is easily understood by professional developers.

The main difference between UML class diagrams and functional type declarations is that the type declarations do not consider operations that will be performed on the data. However, this is not often seen as an issue by attendees of our courses.

3.2 Unions, class hierarchies and extensibility

Although discriminated unions model the same data structures as class hierarchies, the way they can be extended differs. In fact, the two represent two extreme sides of the Expression Problem [13]. Although this is an intriguing problem, it is mainly of interest to programming language researchers, so we focus on explaining the two extreme sides: object-oriented and functional.

For our audience, the object-oriented version using virtual methods is well-known. We focus on explaining the functional version, which can be written using discriminated unions in F#. As this may appear unnatural at first, we clarify the difference by looking at an object-oriented implementation of the solution.

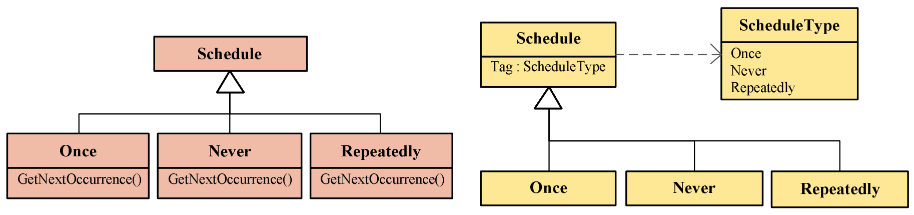

Figure 2. Object-oriented (left) and functional (right) approach extensibility.

The two alternatives, both represented using classes, are shown in Figure 2. In the left version, virtual method `GetNextOccurrence` is defined as part of the base class and overridden in every sub-class. In the version on the right, the base class contains a `Tag` property that determines which of the cases a `Schedule` object represents. The body of `GetNextOccurrence` for the version on the right looks as follows:

```
switch (schedule.Tag) {
  case ScheduleType.Never:
    return DateTime.MinValue;
  case ScheduleType.Once:
    var o = (Once)schedule;
    return o.Date > DateTime.Now ? o.Date : DateTime.MinValue;
  case ScheduleType.Repeatedly:
    // Complex calculation omitted
}
```

The snippet uses `switch` to determine the case using the `Tag` property. When it needs more properties of the sub-type, it uses type cast to extract the properties.

Example summary. The classes and code sample shown above demonstrate how F# compiles discriminated unions. This means that one purpose of the example is to demystify pattern matching on discriminated unions. Moreover, encoding discriminated unions using the `Tag` property is used in .NET libraries for representing LINQ expressions [14] proving that the representation may be useful, even in C#.

Languages like F# and Scala provide language support for both options, so developers need to learn how to choose between them. Our rule of thumb used in F# is to start with a discriminated union, because it provides simpler model of the domain. We recommend the use of class hierarchies only for .NET interoperability or when the need for adding new cases without changing existing code-base is clear (i.e. plugins).

3.3 Transformations and multiple representations

After discussing how to design functional data structures, we turn our attention to data processing. In functional programming, it is common to use multiple different representations of data and transform between them during the execution (for example, different phases in a compiler). The idea of using multiple representations does not have a good analogy in the object-oriented design, so it needs to be explained separately.

For example, the checkout counter might scan the items and build a data structure discussed earlier, but then it would turn it into a structure that represents the final sale – a dictionary containing quantity for every purchased product.

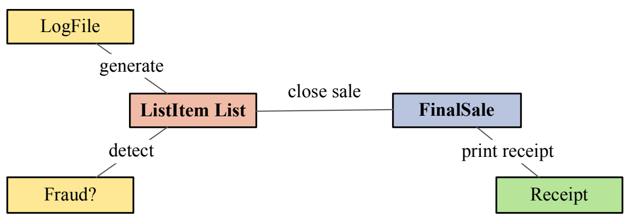

Figure 3. Data transformations in the checkout counter application.

The transformations implemented in the checkout counter are shown in Figure 3. The main operation is “close sale”, which turns `ListItem list` into `FinalSale`. Remaining operations, such as checking for a fraud or printing the receipt are performed on a representation that is more appropriate for the task. This is the main benefit of using multiple representations – once we implement the main transformation most other operations are very easy to add.

3.4 Recursive processing and visitors

Although the idea of using multiple representations is less common in object-oriented style, the code that implements individual transformations can be often related to the Visitor design pattern [3]. Some of the common structures used in discriminated union declarations can be also related to well-known design patterns. Consider the following example inspired by the library for modeling of financial contracts [15]:

```
type Contract =
  | Trade of string * decimal
  | Until of Contract * DateTime
  | Both of Contract * Contract
```

A financial contract is either a primitive `Trade` (consisting of a commodity name and an amount to be traded), or `Until` contract (which limits the validity of another contract), or it is `Both` contract (consisting of two other contracts).

Structural design patterns from object-oriented programming usually aim to represent some structure, but hide it from the users of the type. This is not our concern with discriminated unions, because the structure is transparent. However, the structure we used in the above example corresponds to two standard design patterns. The `Both` case combines multiple different contracts, which is similar to the Composite pattern and the `Until` case decorates another contract with additional information or functionality, which is what the Decorator pattern does.

However, the main point of this section is to discuss the processing of recursive types, such as the `Contract` discriminated union. For simplicity, we look at code that prints trades that can happen at a given day, but the situation is the same when implementing transformations between representations (as discussed in Section 3.3):

```
let rec run day = function
  | Trade(stock, amount) -> printfn "%s (%A)" stock amount
  | Until(contr, limit) -> if day < limit then run day contr
  | Both(left, right) -> run on left; run on right
```

As already mentioned, discriminated unions correspond to class hierarchies. We can add new operations, without modifying the type declaration (adding abstract methods). This can be achieved by adding the `Tag` property (which is what the F# compiler does), or more cleanly using the Visitor pattern [3]. When following the pattern, we would define a base class `ContractVisitor` (with abstract method for every case). To implement processing, such as the `run` function, we add a new subclass of the visitor (`RunContractVisitor`) and implement behavior in the virtual methods.

Example summary. To a functional programmer or a first-year student, the relation between advanced object-oriented concepts, such as Composite, Decorator or Visitor and simpler functional ideas would not be very useful. However, to developers who think about problems in terms of these concepts, the relation provides a useful mental model. They can first design functional programs by translating the OO patterns to functional concepts. As they become more familiar with the functional style, they gradually start thinking about problems using, arguably simpler, functional terms.

The relation between the Visitor design pattern and a recursive function over a discriminated union is particularly good motivation for learning functional programming. The amount of code required to implement Visitors is incomparable. At this point, it is also worth mentioning the discussion whether design patterns are just workarounds for missing language features [16].

In this section, we only sketched the relationship between the OO design patterns and functional ideas. More details can be found in Section 7.5 of our book [12].

4 Learning from functional programming

It is often anecdotally claimed that learning functional programming "makes you a better programmer". Functional programming provides an alternative, often simpler, way of thinking about problems. Moreover, many techniques from functional programming can be successfully applied in other languages.

Learning about functional techniques that could be applied in other languages is often the motivation for readers of Real-World Functional Programming or attendees of our F# courses. Large companies are often reluctant to changes so many attendees will continue writing code in C#. In this section, we discuss two concepts that are best understood in functional setting, but are practically useful in languages such as C#.

4.1 LINQ and first-class values

The C# language version 3.0 introduced LINQ [18] – a set of features that simplify data processing and is heavily introduced by functional concepts. In practice, LINQ is mainly used for the purpose for which it was designed for 9working with collections) and so developers often do not realize the potential of underlying features.

When relating LINQ and F#, we start by looking how to solve data processing tasks in F# using an approach that is similar to a LINQ solution. Then we look at other applications of the same concepts in F#. Finally, we show that the same ideas can be also useful in C# using LINQ.

As an initial example, the following code shows a how to calculate floating 5-day average of stock prices returned by a function `getPriceData`:

```
getPriceData "MSFT"
|> Seq.windowed 5
|> Seq.map (Array.average)
|> Seq.iter (printfn "Price: %f")
```

The example creates a sequence of 5-day windows using `Seq.windowed`, calculates the average for each window and prints the averages. The pipelining operator (`|>`) serves a similar purpose to C# 3.0 extension methods, but it is defined in a library, which shows the flexibility of F#. Functions `Seq.windowed` and `Seq.iter` do not have LINQ counterpart (although they are easy to implement) and they are also not supported by the LINQ query syntax, which shows that composing functions has more uses than just for building queries. Finally, the use of partial function application reduces the amount of code that needs to be written to solve the problem.

The most interesting aspect of the above example is that it can be easily adapted to work on live prices instead of in-memory data. We can use the `Observable` module instead of `Seq` and change the function that loads the data to a live version as follows:

```
getPriceDataLive "MSFT"
|> Observable.windowed 5
|> Observable.map (Array.average)
|> Observable.add (printfn "Price: %f")
```

The code looks very similar, yet it works quite differently. The `getPriceDataLive` function returns an object that emits prices as they become available. When 5 values are generated, the code calculates the average and prints it. As the data source produces new prices, a new floating average is printed. In a standard .NET library, the data source would be exposed as an event. F# treats events are treated as first-class values that can be passed around as values. As a result, we can process them using higher-order function in the style of functional reactive programming [21].

In C#, events are a special language feature and are not treated as first-class values. This makes implementing such library more difficult and, arguably, makes it harder to discover the possibility. Such C# library is now available [19] and it is likely that it has been inspired by the earlier F# design [20]. The library also supports LINQ and makes it possible to write simple live data processing as a LINQ query.

Example summary. The key message of the above example is that F# makes it easier to design innovative libraries that may also benefit from the LINQ query syntax. From a C# perspective, it is more difficult to imagine the possible uses, because object-oriented style does not focus on working with “first-class values”. Representing events as values (instead of special kind of class members) allowed the development of Reactive Framework library.

We can emphasize the point by returning to option values discussed in Section 2.3. Previously, we looked at the implementation of higher-order functions for option values and used them to avoid `null` values. After defining several appropriate extension methods, it is possible to use the following query syntax:

```
from prod in DB.TryFindProduct(id)
from cat in DB.TryFindCategory(prod.CategoryID)
select prod.Name + ", " + cat.Name
```

This use of query syntax may be too subtle for a practical use, yet it is very attractive. Although it demonstrates the flexibility of C# queries and F# computation expressions

[8], it shows another a more important idea. Many different types share similar operations. When teaching functional programming, we emphasize this idea as opposed to explaining monads or details of the query and computation expression syntax.

4.2 Domain specific languages

The idea of embedded domain specific language (DSL) is probably familiar to most functional programmers – given a class of problems, we design a "language" that is suited for solving problems of this class. Embedded domain specific languages are hosted in general purpose languages, but they look as stand-alone languages, usually thanks to the syntactic flexibility and extensibility of the host language.

Designing embedded domain specific languages has a long tradition in functional languages, but it is becoming more popular in languages such as C# or Java [4]. Recent extensions in C# 3.0 make the language more flexible, allowing better integration of domain specific languages. The experience with compositionality and declarative style from functional programming is very valuable when designing DSLs, regardless of the language used for the implementation.

Representation techniques. The first principle from functional programming that developers can use is to hide the internal representation of the DSL. When designing a DSL, we should find a minimal set of primitives that can be used to express any construction in our language. A definition based only on these primitives will not need to be modified if the internal representation of the language changes.

To find the minimal set of constructs, functional programmers would probably define a discriminated union, such as the `Contract` type declared in Section 3.4. However, the constructs would be exposed as functions that hide the internal structure:

```
let trade (what, amount) = Trade(what, amount)
let until dt contract = Until(dt, contract)
let ($) c1 c2 = Both(c1, c2)
```

Although the internal implementation of DSL is hidden, functional style is clearer at distinguishing between two possible representations. It also makes the benefits and drawbacks of the two alternatives more obvious:

- **Data-centric representation.** Using a discriminated union type as above makes it is more difficult to add a new primitive to the language (i.e. choice between two contracts), but it allows using the same value for multiple purposes (i.e. valuation of the contract and its visualization or execution).
- **Behavior-centric representation.** The second option is to represent contracts as a wrapped function that performs a specific task (i.e. return price of the contract). This allows adding new primitive contract types, but it makes it impossible to use the same type for other purposes.

When using object-oriented languages, the two alternatives would be both represented as classes. As discussed in Section 3.2, object-oriented methodologies favor behavior-centric representation (classes with abstract methods), so clearly seeing both alternatives helps developers to choose the right alternative. In fact, the data-centric representation is usually more suitable for domain specific languages.

Syntax and compositionality. The design of domain specific languages in functional language usually has two goals. To provide composable set of primitives and to give a readable syntax that makes the usage look as a special purpose language. For example, a complex financial contract might be modeled as follows:

```
onDate     (DateTime(2012, 4, 30)) (trade ("MSFT", 23.0)) $
repeatedly (DateTime(2012, 4, 23)) (TimeSpan.FromDays(7.0)) 10
           (trade ("AAPL", 220.0))
```

The example demonstrates both aspects. The functions `onDate` and `repeatedly` can be defined in terms of the primitives introduced earlier (`onDate` limits validity using both `until` and `after`, while `repeatedly` generates specified number of occurrences with a given time span and combines them). The syntax of the snippet is free of clutter due to the applicative style and the use of custom operators.

The compositionality of the DSL can be achieved equally in language such as C#. The most convenient syntax can be obtained using method chaining:

```
Trade.Create("MSFT", 23.0).
  OnDate(DateTime(2012, 4, 30)).And(
Trade.Create("AAPL", 220.0).
  Repeatedly(DateTime(2012, 4, 23), TimeSpan.FromDays(7.0), 10))
```

The C# version of the sample is similar to the previous F# version. It is slightly longer (and thus it is formatted differently in the paper), but it has exactly the same structure. When implementing the code, `OnDate` and `Repeatedly` would be provided as extension methods, which means that they can be added by the user of the DSL. The `And` primitive could be implemented as overloaded `+` operator, but C# does not allow defining arbitrary operators, so we instead choose a named method.

Example summary. The main purpose of this example is to demonstrate how functional programming helps with designing domain specific languages. The example teaches several aspects of DSL design in F#, but at the same time, it shows that the same concepts can be used in C#. The example shown here is based on our advanced F# course [17] and follows explanation from Chapter 15 of our book [12].

By implementing the sample first in F#, we can easily explain the most important aspects of DSL design: choosing between data-centric and behavior-centric representation and selecting the set of composable primitives. These concepts are easier to consider in the functional setting (i.e. when writing a discriminated union).

The usage of the DSL in the last snippet also reminds developers of numerous aspects of functional programming. It is written in a declarative style and emphasizes "what" instead of "how". The style also affects more technical aspects of the C# code – the sample is written as a single expression (indeed, functional languages are often expression-based) and it uses minimal amount of language keywords (again, functional languages often need fewer keywords).

5 Conclusions

5.1 Related work

Functional programming has been taught at universities for a long time and it led to the development of influential teaching materials, such as Programming in Haskell [25] and Structure and Interpretation of Computer Programs [22]. Although the latter is acclaimed by many professional developers, it relies on broad knowledge of computer science, which is not common among professionals. This point was also one of the criticisms from the authors of How to Design Programs [23, 24].

More recently, several books focused on explaining functional programming languages to professional developers. Real-World Haskell [26] is perhaps the best example. It uses real-world examples (similarly to our example in Section 3.1), but it focuses on teaching functional programming in Haskell without hinting what ideas may be applicable in daily programming job when using another programming language.

5.2 Summary

In this article, we described our approach to teaching functional programming concepts and the F# language to professional software developers, mainly coming from the C# community. The paper is based on a book Real-World Functional Programming [12] and on commercially offered F# trainings [6].

Our approach is based on two key principles. First, we want to build on the existing knowledge that professional developers have; especially object-oriented programming and design patterns. Second, we do not want to teach just the F# language – we want to explain functional programming concepts that are useful when thinking about programming in general and we also present techniques that are useful in other programming languages.

As discussed in this paper, we start by explaining core aspects of functional programming by relating them to common patterns or pitfalls in object-oriented style. For example, immutability can be related to the Expression Builder pattern, design of functional types can be related to designing class hierarchies using UML and processing of recursive data types can be explained using the Visitor pattern.

We believe that our approach provides the pragmatic methodology for teaching functional concepts to professional software developers that is needed to demystify functional programming and help it succeed in the industry.

Acknowledgements. The author is grateful to Manning Publications and to Jon Skeet who influenced and contributed to the Real-World Functional programming book. The F# trainings have been developed in collaboration with Phil Trelford who contributed several of the ideas presented in this paper.

References

1. M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern (online, 2004). Retrieved May 2012, from: http://martinfowler.com/articles/injection.html
2. M. Fowler, E. Evans. Fluent Interface (unpublished manuscript, 2005). Retrieved May 2012, from: http://martinfowler.com/bliki/FluentInterface.html
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994. ISBN: 978-0201633610
4. M. Fowler. Domain-Specific Languages. Addison-Wesley, 2010. ISBN: 978-0321712943
5. A. Hunt, D. Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 1997. ISBN 978-0201616224
6. T. Petricek, P. Trelford. Functional Programming in C# and F# (course outline). Retrieved May 2012, from: http://functional-programming.net/courses/functional-dotnet
7. B. Hurt. The “Hole in the middle” pattern (unpublished manuscript, 2007). Retrieved May 2012, from: http://tinyurl.com/hole-in-the-middle
8. T. Petricek, D. Syme. Syntax Matters: Writing abstract computations in F#. To appear in pre-proceedings of TFP 2012. See: http://www.cl.cam.ac.uk/~tp322/papers/notations.html
9. G. Kiczales, J. Lamping, A. Mehdhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin. Aspect-Oriented Programming. In Proceedings of ECOOP 1997
10. Microsoft Corporation. C# Language Specification, Version 3.0 (2007). Retrieved May 2012, from: http://tinyurl.com/csharp3-specification-doc
11. Oracle Corporation. JSR 335: Lambda Expressions for the Java™ Programming Language. Retrieved May 2012, from: http://www.jcp.org/en/jsr/detail?id=335
12. T. Petricek with J. Skeet. Real-World Functional Programming: With Examples in F# and C#. Manning, 2009. ISBN 978-1933988924
13. P. Wadler. The Expression Problem (unpublished note). Java-genericity mailing list, 1998. Retrieved May 2012 from: http://www.daimi.au.dk/~madst/tool/papers/expression.txt
14. Microsoft Corporation. Expression Class (MSDN Library). Retrieved May 2012, from: http://msdn.microsoft.com/library/system.linq.expressions.expression.aspx
15. S. P. Jones, J-M. Eber, J. Seward. Composing contracts: an adventure in financial engineering. In proceedings of ICFP 2000.
16. Portland Pattern Repository (Cunningham & Cunningham). Are Design Patterns Missing Language Features? Retrieved May 2012, from: http://tinyurl.com/patterns-missing
17. T. Petricek, P. Trelford. Advanced F# Programming (course outline). Retrieved May 2012, from: http://functional-programming.net/courses/real-world-fsharp
18. Microsoft Corporation. Language-Integrated Query (LINQ), 2008
19. Microsoft Corporation. Reactive Extensions for .NET (Rx), 2010
20. D. Syme. F# First Class Events (unpublished manuscript, 2006). Retrieved May 2012, from: http://tinyurl.com/fsharp-events
21. C. Elliott, P. Hudak. Functional Reactive Animation. In ICFP 1997.
22. H. Abelson, G. J. Sussman, J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, 1996 (2nd ed), ISBN 0262510871
23. M. Felleisen, R. B. Findler, M. Flatt and S. Krishnamurthi. How to Design Programs: An Introduction to Programming and Computing, MIT Press 2001. ISBN: 9780262062183
24. M. Felleisen, R. B. Findler, M. Flatt and S. Krishnamurthi. The Structure and Interpretation of the Computer Science Curriculum. In Journal of Func. Prog., Vol. 14 , Issue 4, 2004
25. G. Hutton. Programming in Haskell, Cambridge University Press 2007. ISBN: 0521692695
26. B. O'Sullivan, D. Stewart, and J. Goerzen: Real World Haskell, O'Reilly 2008, ISBN: 9780596514983