SAMPLE CHAPTER

# F# DEEP DIVES

EDITORS Tomas Petricek • Phillip Trelford

CONTRIBUTORS Chris Ballard • Keith Battocchi • Colin Bull • Chao-Jen Chen • Yan Cui • Johann Deneux • Kit Eason
Evelina Gabasova • Dmitry Morozov • Tomas Petricek • Don Syme • Phillip Trelford

MANNING

# *F# Deep Dives*

by Tomas Petricek
Phillip Trelford

**Chapter 1**

# *brief contents*

# 1 Succeeding with functional-first languages in the industry

Tomas Petricek with Don Syme

Any other programming book would start by introducing the technology it's about, but we're going to do things differently. See, most of the time, you don't read programming books just because you want to learn about a technology. Of course you want to learn the technology, but that's secondary—you want to learn it because you face a practical problem that you need to solve, and you want to do this more rapidly, more reliably, and with fewer bugs than with the technologies you were using before. For that reason, this chapter isn't focused on the F# language, but instead on solving practical business problems.

When talking about programming languages, it's easy to lose this big picture—we programmers are often excited about interesting technical aspects, innovative language features, or elegant ideas. But the evolution of programming languages really does matter in practice, because it enables us to tackle more complex problems and build products that we couldn't even imagine a couple of years ago. For example, who would believe that computers would be able to instantly translate spoken English to spoken Chinese, while maintaining the style of the speaker's voice?[1]

In this chapter, we'll look at the business motivations behind using F#, drawing from a number of case studies made by existing F# users. The technical aspects of many of the case studies are explained in later chapters by the people who developed and successfully deployed them. We'll start with a business situation that many F# users share, and then we'll look at the business problems they faced and how they solved them.

[1] BBC News, "Microsoft demos instant English-Chinese translation," November 9, 2012, www.bbc.co.uk/news/technology-20266427.

But before discussing the main topic of this chapter—business motivations—we'll briefly look at how F# fits in with the current industry trends and at the rich F# ecosystem that combines commercial companies and an enthusiastic open source community.

## F# as part of an ecosystem

Technologies never exist separately in themselves, and F# is no different. From an overall perspective, it fits perfectly with two important trends in the industry: functional and polyglot programming. At a closer look, there's a large and lively ecosystem around F# that's represented by the F# Software Foundation (www.fsharp.org). Let's look at what this means in practice.

### Reflecting industry trends

In recent years, there have been two clear trends in the programming language world:

- *Functional programming* is now undeniably a trend in the industry. An increasing number of programming languages support the functional paradigm, including C++, C#, JavaScript, Python, and Java 8. Moreover, the functional approach underlies many successful libraries, including LINQ and Task Parallel Library (TPL) in the .NET world, and also jQuery and Node.js.
- *Polyglot programming* is the trend of combining multiple languages or paradigms in a single project to take advantage of their benefits where they can be of most use. At the same time, polyglot programming makes it easier to integrate existing stable components with new additions. When using multiple languages, you don't need to rewrite the entire system in a new language when you want to use it—it's perfectly possible to write new components in a different language and integrate them with the existing codebase.

How about F#? First, it's a *functional-first* language. This means F# encourages programmers to use the functional approach, but it fully supports other paradigms. You can use object-oriented style for integrating F# code in larger systems, and you can use imperative style to optimize performance-critical parts of your code.

Second, F# can integrate with a wide range of platforms and languages. It can be compiled to .NET and Mono, and also to iOS and Android (using Xamarin tools) or JavaScript (using WebSharper or FunScript). The type-provider mechanism allows integration with environments like R and MATLAB, as well as databases, WSDL and REST services, and Excel. Let's go a little bit deeper before moving on.

#### MAKING FUNCTIONAL-PROGRAMMING FIRST-CLASS

The About F# page on the F# Software Foundation website has the following tagline:

> *F# is a strongly-typed, functional-first programming language for writing simple code to solve complex problems.*

The *strongly typed* part refers to the fact that F# uses types to catch potential errors early and also to integrate diverse data sources and other programming environments into

the language. As you'll see in later chapters, the types in F# feel different from those in languages like C++, C#, and Java. This is mainly thanks to type inference, which figures out most of the types for you.

The *functional-first* wording refers to F#'s support for immutable data types, higher-order functions, and other functional concepts. They're the easiest way to write F# code, but they're not the only way. As already mentioned, F# supports object-oriented and imperative but also concurrent and reactive programming paradigms.

Finally, the last part of the statement says that F# is a language that lets you solve complex problems with simple code. This is where we need to look at the broader business perspective. We encourage all readers, including developers, to continue reading this chapter; understanding the business perspective will help you succeed with F#.

### MAKING POLYGLOT PROGRAMMING FIRST-CLASS

These days, polyglot programming goes well beyond combining F# and C# on a single .NET runtime. Applications consist of components written in multiple languages, using remote services via REST or WSDL. Scientific computations may call scripts written in R or MATLAB or use optimized FORTRAN or C/C++ libraries; web applications need to call JavaScript libraries; and so on.

As a language that can be compiled to .NET and Mono, F# easily interoperates with languages like C++, C#, and Visual Basic .NET, but that's just the beginning. Without going into the details, here are some of the options:

- *F# on iOS and Android*—Thanks to the Xamarin tools, it's possible to develop iPhone, iPad, and Android applications in F#. The tools come with full F# editor support, based on the community-developed open source MonoDevelop integration.
- *F# for the web and HTML5 apps*—WebSharper is a supported product that lets you develop cross-tier and client-side HTML5 applications using F#. An open source project called FunScript has similar aims and can also import JavaScript libraries using the type-provider mechanism.
- *F# for GPU programming*—F# can be compiled to GPU code using Alea.cuBase. There are also efficient GPU stats libraries like StatFactory FCore.
- *F# and R, MATLAB, and Excel*—F# 3.0 type providers enable integration with R and MATLAB. You can call R and MATLAB functions directly from F# in a typed way with auto-completion. Similarly, you can access Excel data or even run F# in Excel.
- *F# and web-scale data sources*—Type providers bring web-based knowledge to the language. They provide integration with Freebase (a knowledge database), World Bank data, and arbitrary web services and REST-based services.

The type-provider mechanism is explained in chapter 6, and we'll look at how you can write a provider that integrates stock data directly into the language. To understand the balance between different languages in an enterprise context, see chapter 11. For

all the other topics, the F# Software Foundation website (http://fsharp.org) is the best starting point.

Before moving to the main topic of this chapter—the business perspective—let's switch from looking at general industry trends to the ecosystem that exists around the F# language and its commercial and open source contributors.

## Building a healthy environment

F# is an open source, cross-platform language that has a number of industrial supporters as well as a lively open source community. The contributors work together through the F# Software Foundation, which also hosts the F# homepage at www.fsharp.org—a useful resource if you're looking for both technical and nontechnical information about F#.

### F# Software Foundation (FSSF)

To quote the mission statement, "The mission of the F# Software Foundation is to promote, protect, and advance the F# programming language, and to support and facilitate the growth of a diverse and international community of F# programmers." This is achieved in a number of ways.

- FSSF maintains an open source repository for the F# source code and community projects (http://github.com/fsharp), and it manages contributions to key F# projects.
- FSSF seeks to expand the relevance of F# skills and the range of platforms and technologies that can be used with F# and to promote the adoption of F#. This is done, for example, by supporting conferences, training, and other events and collecting testimonials from existing users (http://fsharp.org/testimonials).
- FSSF provides room for affiliated groups, including F# user groups around the world (http://c4fsharp.net) and technical working groups that focus on developing F# in a specific direction, such as data science and machine learning or open engineering.

The F# Software Foundation is registered as a non-profit organization and allows those who agree with the mission statement to join. It also encourages members to join specific technical working groups where they can engage with the community and help to work toward FSSF's goals.

FSSF guarantees long-term support for F# and provides a collaboration platform for all the interested parties:

- MSR Cambridge contributes to the language design.
- The community develops open source extensions and tools.
- Xamarin provides support for iOS and Android.
- The Microsoft product group builds professional F# tooling for Windows.
- SkillsMatter provides F# training and conferences.
- BlueMountain Capital contributes to key data-science libraries.

And this is just the start of the list!

## F# from a business perspective

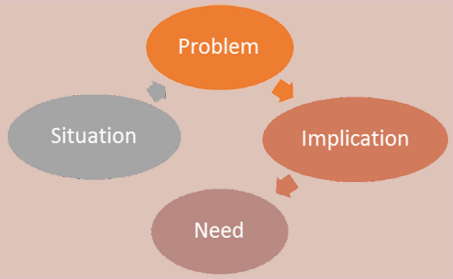
The problem with understanding the business needs for F# (or any other programming language) is that programming languages are complex technologies. Their implications for business are indirect and can be hard to imagine. The "Learning from case studies" section later in this chapter will discuss concrete areas where F# is used, but first let's look at the problem more generally—what are the business motivations for adopting F#?

To deal with this question, we'll borrow ideas from SPIN,[2] which is a methodology for "selling complex products." But don't worry—this isn't a sales-pitch chapter! The methodology tells us that we need to ask four important questions to understand the business perspective for a complex technology. In this chapter, we'll go through some common answers from F# adopters (but, of course, the situation is different for every company).

### SPIN selling

The idea of SPIN selling is to ask potential customers a series of questions that help them understand the business needs for the new technology (as illustrated in the figure):

- *Situation*—What is the customers' existing situation? In our context, what software are they developing, and what are their constraints?
- *Problem*—What problems do the customers face in their current situation? What do they struggle with during the development of their projects?
- *Implication*—What are the business implications of those problems? Do the problems mean projects aren't finished on time or that developers can't deliver products with all the required features?
- *Need*—What is needed to overcome these problems? How can a new technology, such as a programming language, help solve these problems?




**The SPIN selling methodology describes the situation, followed by a specific problem. It proceeds to implications of the problem and only then asks, "What is needed to solve the problem?"**

You can probably imagine a lengthy sales call based on these questions, but also look at the positive side. It's all too easy for a technical person to skip the first and third questions and say something like, "Our trading system doesn't scale, so we need to rewrite it in F#." This might be true, but it's a difficult argument to make without understanding the business context.

[2] Neil Rackham, *SPIN Selling* (McGraw-Hill, 1988).

The business situations for each chapter in this book are different, ranging from companies developing financial systems or models to companies developing user interfaces in games and web applications. One of the most common situations for F# adopters is the development of *analytical* and *data-rich* components.

### Analytical and data-rich components

Most applications contain a computational core that implements business logic, or a component that accesses a wide range of data sources. For some applications (such as CRUD user interfaces), the computational core may be simple, and most of the focus may be on data. For other applications (such as games), data access is minimal, but the computation matters.

Such analytical and data-rich components are what make the application valuable, but with such a general definition, the value may be hard to see. Here are some examples from later chapters:

- Financial models and insurance-calculation engines, such as those discussed in chapters 2 and 4, are examples of analytical components.
- Analytical components in games include artificial intelligence but also the component that's responsible for managing the flow of the gameplay (see chapter 9).
- Another example of an analytical component is an algorithm that analyzes social networks and tells you how to better target advertisements, or recommends people whom you might want to follow (see chapter 5).

So, what's the general business situation we're looking at? For the purposes of this chapter, let's imagine that you're leading a team developing analytical or data-rich components. Other business situations, such as developing complex user interfaces, are equally important, but choosing one scenario will help us keep the chapter focused.

> **TECHNOLOGY RADAR** The choice of analytical and data-rich components as our motivating scenario isn't an arbitrary decision. ThoughtWorks' *Technology Radar* publication recommends exactly this use of F#, although using a different wording: "F# is excellent at concisely expressing business and domain logic. Developers trying to achieve explicit business logic in an application may opt to express their domain in F# with the majority of plumbing code in C#."[3]

The first task is to understand the business problems that you might have as a team leader for a company developing analytical and data-rich components.

[3] ThoughtWorks, *Technology Radar*, March 2012, http://mng.bz/wZvF.

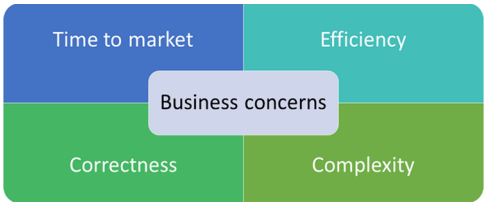



**Figure 1 The key business concerns for developing analytical and data-rich components**

## Understanding business problems and implications

As already mentioned, we're focusing on analytical and data-rich components. Imagine a team developing financial or actuarial models, a team developing server-side components for a massive multiplayer game, or a team building machine-learning algorithms for a social network or ecommerce recommendation system.

There are a number of specific criteria for such systems. An important part of the development process is research or prototyping. Developers need to be able to quickly try multiple, often complex, algorithms and test them. These then have to be deployed to production as soon as possible; and for computationally heavy tasks, the algorithms need to be efficient.

The four most common problems are summarized in figure 1. Analytical applications typically implement more *complex* tasks than the rest of the system; they only deliver value if the implementation is *correct*, is delivered *in time*, and satisfies nonfunctional requirements such as *efficiency*. Table 1 revisits the problems and explores their business implications.

**Table 1 Business problems and their implications**

| | Problems | Implications |
|---|---|---|
| Correctness | As computers become more efficient, financial and actuarial models grow increasingly complicated. The amount of available data grows equally quickly, so algorithms that process this data become more advanced.<br>Maintaining the correctness of such systems raises many problems. It becomes difficult to add new features without breaking existing code, and systems may break as data sources and formats evolve. In settings where models are developed by researchers and are later reimplemented by developers for production code, it's hard to keep the two in sync.<br>An incorrect system that produces incorrect values can easily lead to wrong decisions being made. | If a user interface displays a picture incorrectly, your user will likely be annoyed, but they won't lose money. But money can easily be lost if something goes wrong in an analytical component of a financial system.<br>An infamous example of a correctness problem was the Mars Climate Orbiter probe launched by NASA in 1998. The probe failed during launch because one part of the system was using metric units (force measured in Newtons) and another was using imperial units (measured in pound force).<br>Even when incorrect systems don't have such massive consequences, they may lead to buggy services and the loss of reputation for the company, or to buggy products and a loss of customers. |

**Table 1 Business problems and their implications *(continued)***

| | Problems | Implications |
|---|---|---|
| Time to market | Another important consideration for analytical and data-rich components is the *time to market*—how much time is needed before an initial idea can be turned into production-quality code.<br>For example, a financial company might have a research department that develops models in statistical or numerical environments like R and MATLAB. When a model is designed and tested, it's passed to developers who translate the models to C++ for deployment to production. Such translation can easily take six months.<br>Consider another example from the social gaming domain. A quick release cycle is important to make sure that your players keep getting new features, or even new games, every few weeks. | In the financial sector, the inability to turn a new mathematical model into a system that can be used in production might mean the business loses an opportunity that exists only in a short timeframe.<br>In the social gaming world, a company will quickly lose players if the games aren't rapidly updated or new features aren't added.<br>The time to market is also important in the startup world, which is symbolized by the phrase "fail fast." You want to be able to develop initial prototypes quickly, so that you can immediately verify the viability of some idea. If the prototype does work, you should also be able to quickly turn it into a complete project. |
| Efficiency and scalability | Two related concerns are efficiency and scalability. Efficiency is mainly important for computationally heavy software such as financial models. For example, models that were originally developed by researchers in R or Python need to be translated to more efficient C++ code or optimized Python. If the researchers were able to write their models more efficiently, then the translation step wouldn't be needed.<br>Scalability matters even for software that doesn't perform heavy computations. A server-side application (such as a social game backend) or UI (a game frontend) needs to handle multiple concurrent requests or user interactions. | Efficiency and scalability have varying importance in different contexts.<br>A common case for efficiency in financial systems is that models need to be recalculated overnight, so there's a hard limit. Failure here means up-to-date information isn't available. Similarly, when serving a web page with ads, the ad service needs to choose an appropriate ad based on the user's information almost instantly.<br>As for scalability, server-side code that doesn't scale will consume excessive resources and make maintenance costly. On the client side, nonscalable applications can hang and lead to a poor user experience. |
| Complexity | Analytical and data-centric components are usually the parts of an application that implement advanced logic. They provide value by implementing mathematical models, data analyses, or processing of concurrent events.<br>In a poorly designed system, complexity can easily grow beyond a tractable level, most commonly because different features that should be independent interact in unexpected ways. | As a result of increasing complexity, your company might not be able to implement a desired financial model, AI behavior, or data analytical component, because it's too complex.<br>As a result, you won't be able to provide the data that users and customers need (or not at the required quality), or an entire project or product may fail.<br>In other words, without the right tools, you'll often have to settle for a suboptimal solution. |

The business problems and implications outlined here are by no means complete, and they overlap. Handling efficiency or complexity often impacts time to market—you need to spend more time optimizing your system or tracking bugs. Efficiency and scalability are also often linked to correctness. In an attempt to make code more efficient, you could easily introduce bugs when trying to parallelize code that uses shared state.

The key takeaway from this section is that developing software is hard. Exactly where the difficulties lie will depend on the particular software you're developing. Understanding these difficulties and how they affect the business is crucial to finding the right way to tackle them, and one solution may be using a more appropriate programming language!

## Inferring business needs

Many of the business problems discussed in the previous section are directly addressed by language features in functional-first programming languages or by their inherent aspects. We'll discuss F#-specific features, but many of these observations apply to other functional-first languages.

### Functional-first programming languages

We use the term *functional-first* to distinguish between purely functional languages and those that combine functional aspects with other paradigms. As with any language classification, this is, to a large extent, a subjective measure.

In *traditional functional languages*, such as Miranda, Haskell, and ML, the only way to write programs is to fully adopt a functional style. There may be some exceptions (such as effects in ML), but the overall program structure has to be functional.

In *functional-first languages*, the functional approach is highly encouraged. This is done mainly by choosing the right defaults and using syntax that makes the functional style more convenient. But it's still possible to use other styles.

In languages like F# and Scala, you can write fully object-oriented and imperative code. This is sometimes needed for interoperability or efficiency—and it's an important aspect that makes such languages successful in practice. But most new code can be written in the functional style and can benefit from properties that make functional languages unique, encouraging correctness, a shorter time-to-market period, efficiency, and the ability to solve complex problems.

Let's now go over the four business problems again, but this time using a more developer-focused perspective. You'll see that functional-first languages have unique features that help you tackle the problems just discussed.

## Writing correct software

As you'll see in the next section, a surprising number of F# users report that they can write software without any, or with a minimal number of, bugs. Where does this result come from? This is a difficult question to answer, because correctness follows from the nature of functional languages. The best way to understand it is to look at two concrete examples.

### LIVING WITHOUT NULL REFERENCES

Tony Hoare introduced `NULL` references in 1965 in Algol. In a 2009 talk, he referred to this language feature as his "billion dollar mistake."[4] If you do a quick search through your bug database, you'll understand why. The number of bugs caused by `NULL` references is astonishing. To tackle this, F# doesn't allow you to use `NULL` with types defined in F#. When you want to represent the fact that a value may be unavailable, you have to do so explicitly using the `option` type.

Compare the following F# and C# code that finds a product by ID and prints its name:

**F# code**

Optional type indicates that a product may not be found

```
let findId id : option<Product> =
  // Find product and return Some(p)
  // or return None if id is unknown

match findId 42 with
| Some p -> printf "%s" p.Name
| None -> printf "Not found!"
```

Compiler checks that all cases are handled

**C# code**

Type doesn't tell you if the function always returns a valid product

```
Product FindId(int id) {
  // Find product or return 'null'
}

Product p = FindId(42);
Console.Write(p.Name);
```

You can access properties without checking for nulls.

Of course, the two snippets are different, and the C# version doesn't correctly handle `null` values. The point is that there's no indication that it should. The type `option <Product>` in F# is an explicit indication that the function may not return a value and that you have to handle that case. In other words, F# allows you to make invalid states non-representable. If you have a `Product` value, then you *always* have a product.

This ability to track information in types goes well beyond `NULL` values.

### CALCULATING CORRECTLY USING UNITS OF MEASURE

Another area where F# uses types to guarantee correctness of code is numerical calculations. In the business implications discussed earlier, we mentioned the Mars Climate

[4] Tony Hoare, "Null References: The Billion-Dollar Mistake," QCon London, August 25, 2009, http://mng.bz/l2MC (video).

Orbiter. The project failed because one part of the system represented force in metric units and another used imperial pound force.

In F#, you can annotate numeric values with information about units, and the compiler ensures that units are used in a consistent way. Consider this F# Interactive session (with the F# Interactive output typeset in italic):

```
> let force1 = 15.0<kg m/sec^2>
  let force2 = 5.0<lbf>
;;
val force1 : float<kg m/sec ^ 2> = 15.0
val force2 : float<lbf> = 5.0
```

**The two numerical values have different unit annotations and thus different types.**

```
> force1 + force2;;
error FS0001: The unit of measure 'lbf' does not
match the unit of measure 'kg m/sec ^ 2'
```

**Adding two numbers with incompatible units generates a compile-time error.**

```
> [<Measure>] type N = kg m/sec^2;;
(...)
```

**Defines derived unit to make the code more readable**

```
> force1 + force2 * 4.4482<N/lbf>;;
val it : float<kg m/sec ^ 2> = 37.241
```

**Compiler checks that multiplication results in a value in Newtons** ❶

The purpose of this example isn't to show that F# is a perfect language for writing control systems for NASA probes. Units of measure can be used in ordinary business applications. For example, you can use them to distinguish between product prices (by defining *dollar*, or a *currency* unit) and ratios that are unit-less.

Also, the example demonstrates a more fundamental and prevailing aspect of statically typed functional-first languages: they use types to capture more about the domain that you're working with. In F#, this is enhanced by sophisticated type inference, so you don't usually have to write the types yourself. In this example, the compiler infers that the result has the `kg m/sec^2` unit (or, in other words, is in Newtons) ❶.

It may seem that thinking about types and missing values will make the development process slower, but this isn't the case. You may need to spend more time when writing the initial implementation, but you'll spend much less time tracking bugs. Moreover, there are a number of other aspects that make the development process faster.

## Reducing time to market

In the previous section, we discussed a scenario where a company uses mathematical software such as R or MATLAB to develop and test models, but then rewrites the software using another language, such as C++, for deployment. F# has also been dubbed "the language that both scientists and developers understand," which makes it a great language for such cases.

Another important aspect of F# is that it interoperates extremely well with the outside world. As a .NET language, it can access a wide range of .NET libraries and easily call efficient C/C++ code when needed. For other environments, such as R or MATLAB, the type-provider mechanism allows smooth integration. Let's first look at how F#

connects the research and development phases. Then we'll quickly look at the integration options.

### FROM RESEARCH TO PRODUCTION

Developers and researchers often have different needs. As a researcher, you want to be able to quickly and easily load data from a local file (without worrying about deployment, because you're running the script locally). Then you want to run some analysis of the data, write algorithms to work with the data, and visualize the results. You don't need to worry about the structure of the code—you're just writing scripts that you'll probably rewrite when you realize that you need another algorithm.

On the other hand, as a developer, you need to create a project and package your algorithms nicely in types and modules that will be easily usable from F# or C#. You need to write unit tests for the algorithm and make sure the library can be integrated with an automated build system.

These two sets of requirements are different, but F# can be used both ways: as an interactive environment and also as a full programming language. This combination makes it possible to smoothly transition from the research phase to the developer phase. The typical process is outlined in figure 2.

### INTEROPERATING WITH R

Another important aspect of F# that helps speed up the time to market is that it can interoperate with existing solutions. If you have an efficient library that does a calculation in C++, you don't want to rewrite it from scratch in another language and spend weeks optimizing it. Similarly, if you have a mathematical model or a useful function written in mathematical package like R or MATLAB, you want to be able to reuse such code and call it from your newly written code.

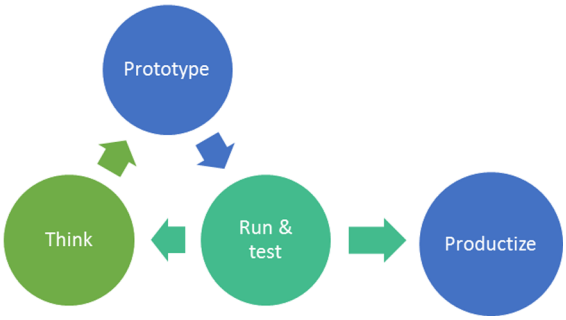



**Figure 2 In a typical R&D process, you start with an interesting idea. F# makes it easy to translate the idea into an initial prototype that is a rough and incomplete implementation but that gives you enough so that you can quickly test it and see if the idea works. This is repeated a number of times until you reach a solution that works—at that point, the existing F# code can be cleaned up, documented, and properly tested and encapsulated, and it becomes part of the product.**

As a .NET language, F# can easily interoperate with any .NET libraries and call C/C++ code through P/Invoke. But the type-provider mechanism takes these integration capabilities much further.

> **F# TYPE PROVIDERS** In a few words, type providers make it possible to write lightweight plug-ins for the F# compiler that generate types based on external information. A type provider can look at a local database and generate types to represent tables, and it can also analyze code in other languages like JavaScript and MATLAB and generate types for interoperating with such code. We'll look at implementing a type provider in chapter 6.

The following brief snippet demonstrates how the R type provider works. Understanding the details isn't important for the purpose of this chapter—the key thing is that you can easily combine data processing in F# and a call to an external function. Given a value `prices` that represents stock prices obtained earlier, you can pass the list easily to the `var` and `mean` functions that are available in the base package in R:

```
#r "RProvider.dll"
open RDotNet
open RProvider
open RProvider.``base``

let divs =
  [ for prev, next in Seq.pairwise prices ->
      log (next / prev) ]

let var = R.var(divs).AsNumeric().First()
let mean = R.mean(divs).AsNumeric().First()
```

**References the R type provider**

**R packages, such as base, become automatically available, and you can see them in autocomplete lists.**

**Calculates a log of differences between consecutive prices in F#**

**Calls an external library via automatically provided functions R.var and R.mean**

This example is slightly artificial, because there are F# functions for calculating variance and mean. But it demonstrates the idea—if you have a more complex function written in another language, the type-provider mechanism allows you to call it easily. An important aspect that can't be shown well in print is that you also get full development environment support. When you type `R` followed by a dot, the editor shows you a list of all available functions (based on the information provided by the R type-provider).

### F# for data science and machine learning

One area where F# stands out is data science and machine learning. In fact, one of the first success stories for F# was an internal system created at Microsoft Research, which implemented a more sophisticated algorithm for ranking ads for Bing.

This is further supported by a wide range of libraries:

- *Deedle* is a library for manipulating structured data and time-series data. It supports grouping, automatic alignment, handling of missing values, and pretty much all you need to clean up your data and perform data analyses.

***(continued)***

- *F# Data* is a collection of type providers for accessing data from a wide range of data sources, including CSV, XML, and JSON files, as well as some online services such as the World Bank or Freebase.
- *F# Charting* makes it possible to create charts interactively with just a few lines of F# code, including multiple charts combined in a single chart area.
- *R type provider and MATLAB type provider* make it possible to call the two most widely used mathematical and statistical packages from F#.

In this section, we mostly focused on scientific and data analytical applications—F# is an excellent fit for this domain, and it's easy to demonstrate how it can help get your ideas to market more quickly. But the "Learning from case studies" section of this chapter looks at a number of case studies, and you'll see that time to market is an important aspect in pretty much every domain where F# is used. Before doing that, let's look at the next two business problems.

## Managing complexity

Can a programming language make it easier to solve complex problems? The case studies on the F# Foundation website show that the answer to this question is yes. In this section, we'll look at a simple example that is explained more in chapter 3. You can also find additional experience reports in the "Learning from case studies" section, later in this chapter.

The key to managing complexity is finding the right abstractions for talking about problems. For example, when describing 3D models, you don't want to do that in terms of pixels—you want to use a language based on triangles, or perhaps basic 3D objects such as cubes and spheres.

This approach is sometimes called *domain-specific languages* (DSLs). The key idea is that you define a language that can easily be used to solve multiple instances of problems of the same class (such as creating various 3D objects or modeling prices of stock options). In functional languages, this concept is so prevalent that you can think of any well-designed library as a DSL.

### PARSING MARKDOWN WITH DSLS

To demonstrate how F# lets you define appropriate abstractions for solving problems, we'll look at *pattern matching*. This is a fundamental idea in functional languages, where you specify rules for recognizing certain data structures (such as non-empty lists, lists starting with a date that's greater than today, and so on). F# makes the mechanism even more powerful by supporting *active patterns* that let you specify custom recognizers for such structures.

In chapter 3, we'll implement a parser for the Markdown document format. A document can contain formatting such as `*hello*` for italicized text or `[F#](http://fsharp.org)` for hyperlinks. Even without understanding the syntax, you can see that the following snippet encodes these two rules:

```
match chars with
| Bracketed '*' '*' (body, rest) ->
    // Create block of italicized text
| Bracketed '[' ']' (body, Bracketed '(' ')' (url, rest)) ->
    // Create hyperlink with body and url
| _ ->
    // Handle other cases
```

**Case when chars starts with a substring containing a body bracketed with asterisks and followed by rest**

**Case when chars starts with a body bracketed using [...], followed by a URL delimited using (...) and then the rest of the text**

The details of the snippet, as well as F# active patterns, are explained in chapter 3. The point of showing this example here is that there's a clear correspondence between our earlier description of the format and the code you write to implement it.

This is why functional-first languages can be so effective at managing complexity—they let you build abstractions (such as the `Bracketed` pattern) that you can then use to elegantly solve the problems at hand (such as recognizing formatting commands for italicized text and hyperlinks).

## Writing efficient and scalable software

The last problem that can have a serious impact on your business is efficiency and scalability. We outlined the implications earlier—financial models need to complete in overnight batches, server-side applications need to handle an increasing number of concurrent clients, UIs should react promptly without blocking, and so on.

This group of problems is diverse. Most important, you need to distinguish between computationally intensive tasks (such as evaluating financial models) and tasks where the main issue is coordinating multiple requests and avoiding blocking.

### COMPUTATIONALLY INTENSIVE TASKS

For computationally intensive tasks, F# provides an excellent balance between code that's easy to write and that runs efficiently. This means initial prototypes can be turned into production-quality implementations without complete rewrites.

This topic is too broad to be covered in a single section, so we'll just summarize the key observations here:

- *Strong typing and generics*—F# is a strongly typed language, which means calculations, array manipulations, and other primitive operations are compiled to native code (although this happens at startup using just-in-time (JIT) compilation). Moreover, the .NET approach to generics means none of this efficiency is lost when working with reusable types such as collections.
- *Performance profile similar to C#*—In general, most programs written in F# will have the same performance profile as C# programs. There are some differences: F# has a number of features, such as inlining, that can make F# code faster, and some functional constructs need to be used with care.
- *Parallelism*—As a .NET language, F# has full access to highly optimized libraries for parallel programming, such as the Task Parallel Library (TPL). Using these from F# is significantly easier than from imperative languages, thanks to the emphasis on immutable data types.

- *Ability to write better algorithms*—Perhaps more important, thanks to the language's expressivity, you can implement more complex algorithms. As a result, you can use more efficient (but complex) models and algorithms.

In summary, F# is efficient enough for any task that you could solve in C#. In many cases it will be faster, either because you can use some nice F# feature or, more important, because it allows you to write more sophisticated algorithms. But for highly optimized computations, you can easily call native C/C++ routines.

#### COORDINATION-INTENSIVE TASKS

In many domains, systems don't need to perform computationally intensive computations, but they do need to handle a large number of interactions, updates from other services, or concurrently running tasks. Consider two sample applications that inspired chapters 8 and 10:

- A trading system that concurrently receives updates from the stock exchange, or other such data sources, needs to display the current state on the screen, perform some simple visualizations, and display them and handle user interaction.
- The server side of a social game only needs to handle service calls from the client side, but it needs to do that for an enormously huge number of requests.

These problems don't just require better libraries or better locking primitives. Writing such applications using traditional concurrency primitives like threads or even tasks won't work because such applications have completely different interaction patterns (see figure 3):

- *Sequential model*—In traditional sequential applications, there's a single control flow. The application has one entry point that calls other components (objects or procedures), and the control proceeds until the result is returned.
- *Parallel model*—In parallel systems that use multiple threads or tasks, the control flow can be forked and then joined to perform more computations in parallel. Nevertheless, the coordination model is still similar to the sequential model.
- *Agent-based model*—The behavior of true concurrent systems is different. It's also called *agent-based*, because the system consists of multiple agents that can communicate by exchanging messages. The system can be structured as agents that solve individual aspects of the problem, like communicating with a stock exchange, handling user-interface requests, or keeping track of some state.

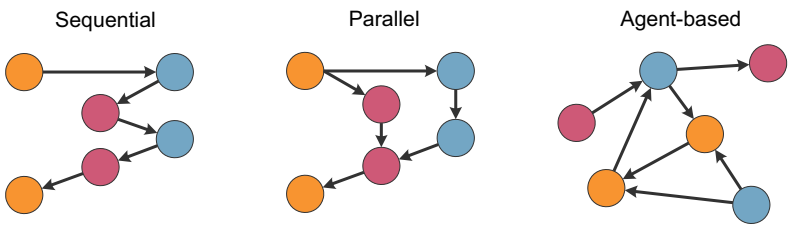



**Figure 3 The control flow in sequential, parallel, and agent-based applications**

The agent-based programming model (also called the *actor model*) is powerful for writing highly concurrent applications. One of its early adopters was Ericsson (using the Erlang language), which still uses it for many applications in telecommunication. The model is directly supported by F# libraries, and it has been used in many major software systems. We'll look at some of them, as well as other case studies, in the next section.

## Learning from case studies

In the preceding three sections, we presented the business aspects of developing analytical and information-rich components. We started by looking at the business situation—what analytical and information-rich components are, and why they're important. Then we looked at common business problems associated with their development. We identified the four most common problems: complexity, efficiency, time to market, and correctness.

> **F# TESTIMONIALS** The F# Software Foundation collects user testimonials and links to detailed case studies at http://fsharp.org/testimonials. At the time of writing, there are some 50 testimonials from a wide range of domains, from DNA programming and modelling to financial and actuarial applications to games and teaching.

Then, in the more technical "Inferring business needs" section, we examined how F# can help in solving these business problems. As a last step, let's now examine existing F# systems and see how they use F# features to solve the problems outlined earlier.

### Balancing the power-generation schedule

The first application that we'll look at is a system developed by a large UK-based energy company. The following is a brief summary of the system:

> *I have written an application to balance the national power-generation schedule for a portfolio of power stations to a trading position for an energy company. The client and server components were in C#, but the calculation engine was written in F#.*

The first important observation is that being able to interoperate with other .NET components is of great importance in this case. The F# portion of the code is relatively small (when measured by line count), but it implements the important and unique logic of the application. This exactly matches the "analytical component" pattern that we talked about earlier.

Now, let's look at three technical aspects that directly match the business problems of developing analytical components:

> *Working with script files and [F# Interactive] allowed me to explore the solution space more effectively before committing to an implementation ….*

This confirms our earlier discussion about integrating research and development. The ability to prototype solutions quickly (and fail fast if an approach doesn't work) is one of the main reasons behind reduced time to market.

The developer next states,

> *The equations I implemented … dealt with units of time, power, and energy. Having the type system verify the correctness of the units of the inputs and outputs of functions … eradicates a whole class of errors that previous systems were prone to.*

Units of measure address *complexity* and *correctness*. By annotating numeric values with their units, it's easier to write complex computations, because you get immediate feedback from the editor about the units of values that you work with. This leads us to the next point:

> *I can be … trying hard to get the code to pass the type checker, but once the type checker is satisfied, that's it: it works. … Weird edge-case errors are minimized, and recursion and higher-order functions remove a lot of bookkeeping code that introduces edge-case errors.*

The author may be exaggerating slightly, but he has good reasons for making such claims:

> *I have now delivered three business-critical projects written in F#. I am still waiting for the first bug to come in.*

Of course, when developing applications in the functional style, you still need to test your code properly. We'll discuss this topic in chapter 12. But it's much easier to test software when the compiler already provides many guarantees, such as the lack of `NullReferenceException` errors.

### Analyzing data at Kaggle

The next interesting user of F# is Kaggle, which is a Silicon Valley based start-up. Kaggle builds a platform for data analysis based on crowdsourcing. Companies and individuals can post their data to Kaggle, and users from all over the world compete to provide the best models:

> *We initially chose F# for our core data analysis algorithms because of its expressiveness. We've been so happy with the choice that we've found ourselves moving more and more of our application out of C# and into F#. The F# code is consistently shorter, easier to read, and easier to refactor, and, because of the strong typing, it contains far fewer bugs.*

Again, the initial usage of F# at Kaggle was for developing analytical components that process data. The company's experience confirms that the choice helped with

correctness (fewer bugs) and time to market (shorter and easier to read code). As in the previous case study, the fact that F# interoperates well with the rest of the world is important:

> *The fact that F# targets the CLR was also critical—even though we have a large existing code base in C#, getting started with F# was an easy decision because we knew we could use new modules right away.*

An interesting observation that Kaggle makes about the use of F# is related to DSLs. We talked about them when looking at the Markdown parser in the earlier "Managing complexity" section. This is what the Kaggle developers have to say:

> *As our data analysis tools have developed, we've seen domain-specific constructs emerge naturally; as our codebase gets larger, we become more productive.*

The essence of this quote is that the developers were able to use the right abstractions without them interacting in unexpected ways. This is perhaps the most important aspect of the functional style for general-purpose programming. By providing the right means for building abstractions, a codebase can naturally grow and can be extended with more orthogonal useful functions; these can be composed to more easily solve problems from the domain of the system.

For our last case study, we'll turn our attention from data analytics to concurrency and gaming.

### Scaling the server side of online games

GameSys is a company developing social, massively multiplayer online role-playing games (MMORPGs). The social gaming domain is unique in that games need to be developed and released extremely frequently. At the same time, the server side needs to be able to scale well—if a game becomes popular, this will cause a massive peak.

At GameSys, F# is used for both the implementation of some small analytical components (that define the behavior of a game) and, more interestingly, for efficient handling of concurrent requests:

> *F# first came to prominence in our technology stack in the implementation of the rules engine for our social slots games, which by now serve over 700,000 unique players and 150,000,000 requests per day at peaks of several thousand requests per second. The F# solution offers us an order of magnitude increase in productivity … and is critical in supporting our agile approach and bi-weekly release cycles.*

> *The agent-based programming model … allows us to build thread-safe components with high-concurrency requirements effortlessly … These*

> *agent-based solutions also offer much-improved efficiency and latency while running at scale.*

Again, F# is used for two of the reasons that we discussed earlier. The first is increased productivity, which leads to shorter time to market. Because of the pressure for frequent game updates, this increased productivity can be a key business factor.

In addition to implementing game logic, the next aspect is building server-side components that handle coordination between computations and distribution of state. The server side of a social game is a typical example of coordination-intensive computation, as discussed in the earlier "Writing efficient and scalable software" section. We won't attempt to explain here why the agent-based programming model is so powerful; you can find the details in chapter 10, which is written by the author of the preceding quote.

## Summary

In this chapter, we focused on topics that are probably more important than code—the business context in which software is developed. Our goal was to give you a conceptual framework that you can use to analyze business problems. When you read the later chapters in this book, you can remind yourself of the four business problems we discussed: time to market, complexity, efficiency, and correctness.

With these four keywords in mind, you'll see that each of the later chapters deals with solving an important business problem, and applying the ideas explained there will have a direct effect on your business. If you're a software developer interested in F#, we hope this chapter also gives you some useful hints that you can use to explain the benefits of F# not just to your technical colleagues, but also to your management.

## About the authors

This chapter is based on Don Syme's talk, "Succeeding with Functional-First Programming in Industry," at NDC 2013.

**Don Syme** is a Principal Researcher at Microsoft and an F# community contributor. He is an advocate of data-rich, functional-first programming with a focus on simplicity, correctness, and robustness of code. Over the last 14 years at Microsoft, he's helped drag programming kicking and screaming into the modern era through technical contributions such as generics for C#, async programming in F# 2.0 and C# 5.0, and type providers in F# 3.0.

**Tomas Petricek** is a long-time F# enthusiast and author of the book *Real-World Functional Programming* (Manning, 2010), which explains functional programming concepts using C# 3.0 while teaching F# alongside. He is a frequent F# speaker and does F# and functional training in London, New York, and elsewhere worldwide.

Tomas has been a Microsoft MVP since 2004, writes a programming blog at http://tomasp.net, and is also a Stack Overflow addict. He contributed to the development of F# during two internships at Microsoft Research in Cambridge. Before starting a PhD at the University of Cambridge, he studied in Prague and worked as an independent .NET consultant.

PROGRAMMING

# F# DEEP DIVES

EDITORS Tomas Petricek • Phillip Trelford

F# is an elegant, cross-platform, functional-first programming language. With F#, developers create consistent and predictable programs that are easier to test and reuse, simpler to parallelize, and less prone to bugs. The language, its tooling, and the functional programming style have proven effective in many application areas like secure financial engines, machine learning algorithms, scientific calculations, collaborative web applications, games, and more.

**F# Deep Dives** is a selection of real-world F# techniques written by expert practitioners. Each chapter presents an important use case where you'll solve a real programming challenge effectively using F# and the functional-first approach. Not only will you see how a specific solution works in a specific domain, but you'll also learn how functional programmers think about problems, how they solve them, and how they integrate F# into existing systems and environments.

## What's Inside

- Covers many subjects including:
  - Numerical computing
  - Data visualization
  - Business logic
  - Domain-specific languages
- Practical solutions to real problems
- Information-rich programming, including LINQ and F# type providers
- Covers F# 3.1 and VS 2013

Readers should have at least an introductory knowledge of the F# language.

**Tomas Petricek** contributed to the development of the F# language at Microsoft Research. **Phil Trelford** is an early adopter of F# and one of its most vocal advocates. They are joined by F# experts **Chris Ballard, Keith Battocchi, Colin Bull, Chao-Jen Chen, Yan Cui, Johann Deneux, Kit Eason, Evelina Gabasova, Dmitry Morozov,** and **Don Syme**.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/FSharpDeepDives

Free eBook
SEE INSERT

"Finally! A book that shows a wide variety of applications for F#."
—Jonathan DeCarlo
Bentley Systems, Inc.

"Beautifully written by F# experts — take a deep dive without holding your breath."
—Kostas Passadis, IPTO

"Outstanding real-world examples that are sure to appeal to both the novice and expert."
—Jeff Smith
ITT Education Services

"I love the *Deep Dives* concept. This book is full of insights about how to apply the power of F# to real-world problems."
—Dennis Sellinger, Géotech SARL

ISBN 13: 978-1-617291-32-6
ISBN 10: 1-61729-132-3
54999
9 781617 291326

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]