Domain-specific languages for finance in F#

Tomas Petricek

http://tomasp.net

More F# in London

Advanced F# by Tomas Petricek & Phil Trelford
2 DAY COURSE. Featuring Phil Trelford
London, Monday, June 11th

Tomas & Phil's Func. Programming in C# and F#
2 DAY COURSE. Featuring Tomas Petricek
, Thursday, September 20th

http://skillsmatter.com | tomas@tomasp.net

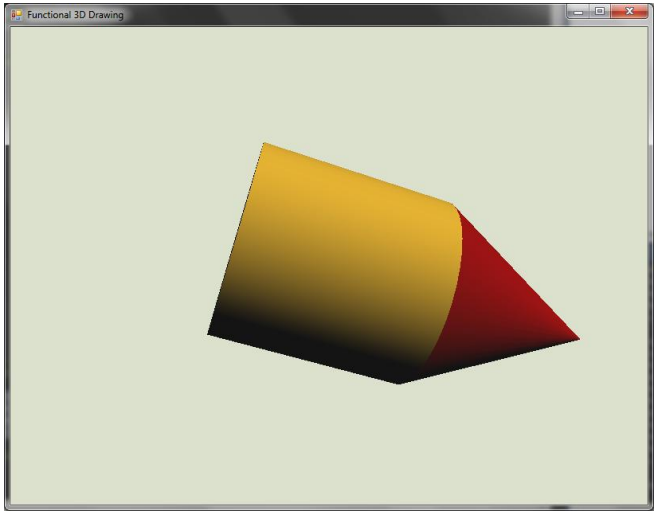
What are **domain-specific languages** and why?

Domain-specific languages

Language for solving **specific problems**

```
Fun.cube
```

```
Fun.cylinder
|> Fun.translate (0.0, 0.0, 1.0)
|> Fun.color Color.Goldenrod $
Fun.cone
|> Fun.color Color.DarkRed
```



Contrast with **general purpose languages**

Domain-specific languages

We have a class of problems

- **Create a language** for the class
- **Use language** to solve them

Functional languages

- **Internal DSL** is just an **F# library**
- **Flexible syntax** and **type checking**
- **External DSL** is a stand-alone langauge

First Example: Modeling financial contracts

Modeling Financial Contracts

Write **contracts** using **simple primitives**

```
let itcontract =
    sellOn
      (DateTime(2012, 4, 30)) ("MSFT", 23.0) $
    purchaseRepeatedly
      (DateTime(2012, 4, 23))
      (TimeSpan.FromDays(7.0))
      10 ("AAPL", 220.0)
```

What can happen on a specific date?

Valuation and risk assessment

DEMO: MODELING CONTRACTS

http://fssnip.net/bJ

Simplifying the example

Date **after** and **until** specified

```
let onDate dt contract =
  after dt (until dt contract)
```

Repeatedly compose trades

```
let repeatedly start span times contract =
  [ for n in 0 .. times ->
      let offs = TimeSpan(span.Ticks * int64 n)
      onDate (start + offs) contract ]
  |> Seq.reduce ($)
```

What are the **primitives**?

Captured as **discriminated union**

```
type Contract =
  | Trade of string * float
  | Opposite of Contract
  | After of DateTime * Contract
  | Until of DateTime * Contract
  | Combine of Contract * Contract
```

Can be **processed** in multiple ways

Only handle **five cases**!

DEMO: PROCESSING CONTRACTS

Domain-specific languages

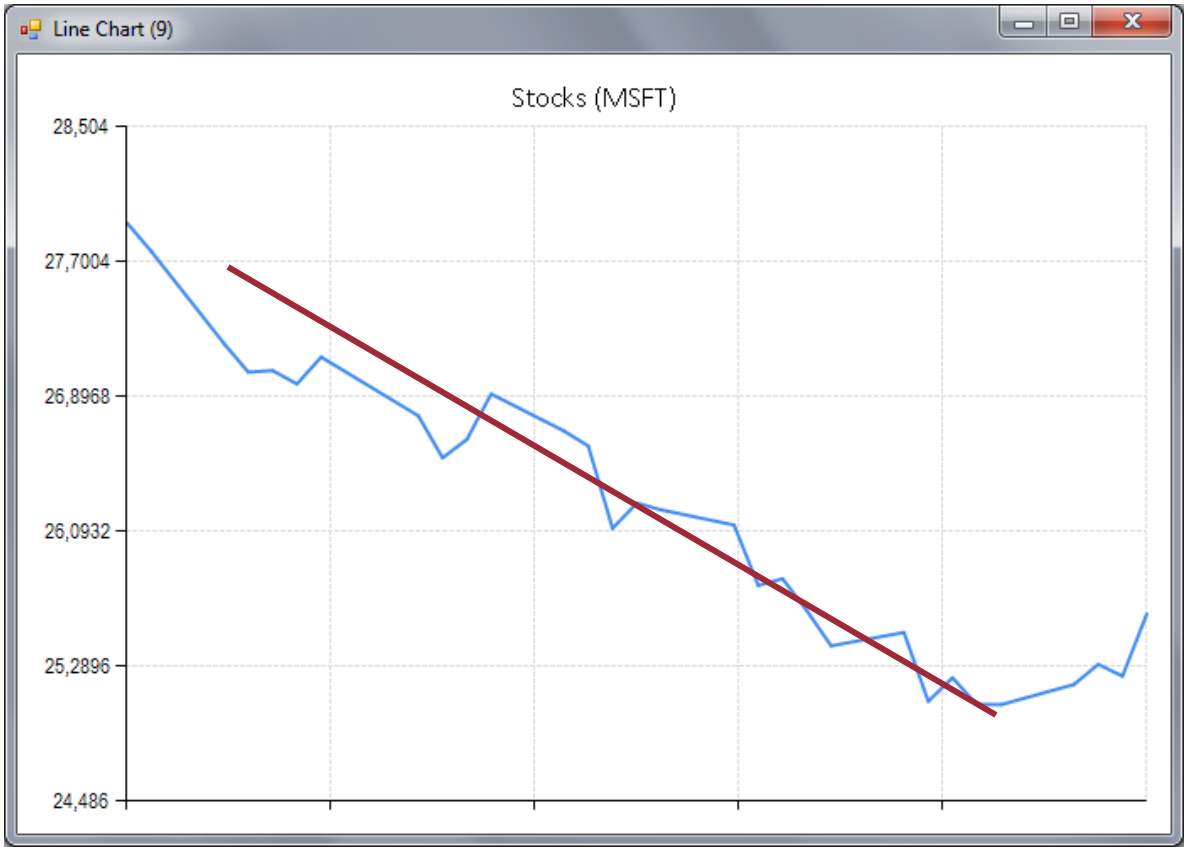
Using **discriminated unions**

- Defines the **language**
- Limits the **expressivity**
- Simplify **common** uses
- Multiple **processing** functions

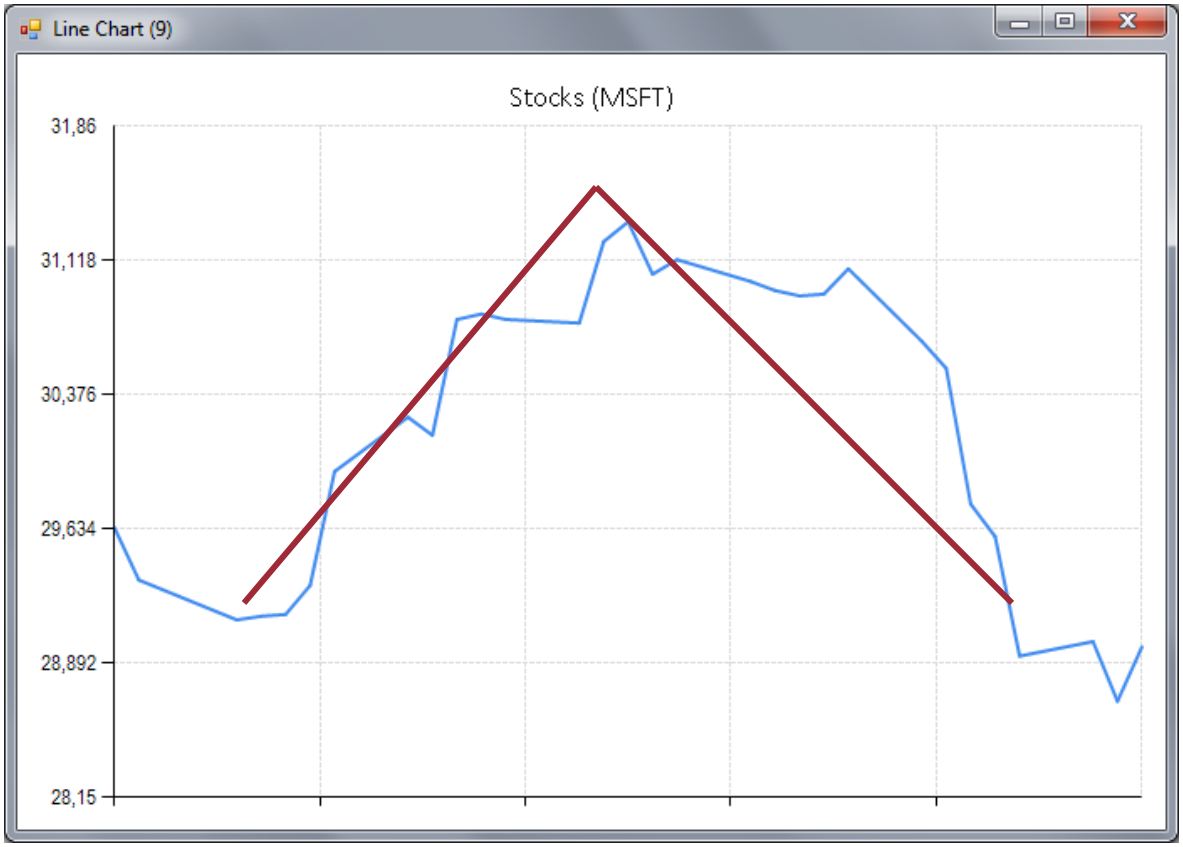
Using **function values**

Second Example: Detecting patterns in price changes

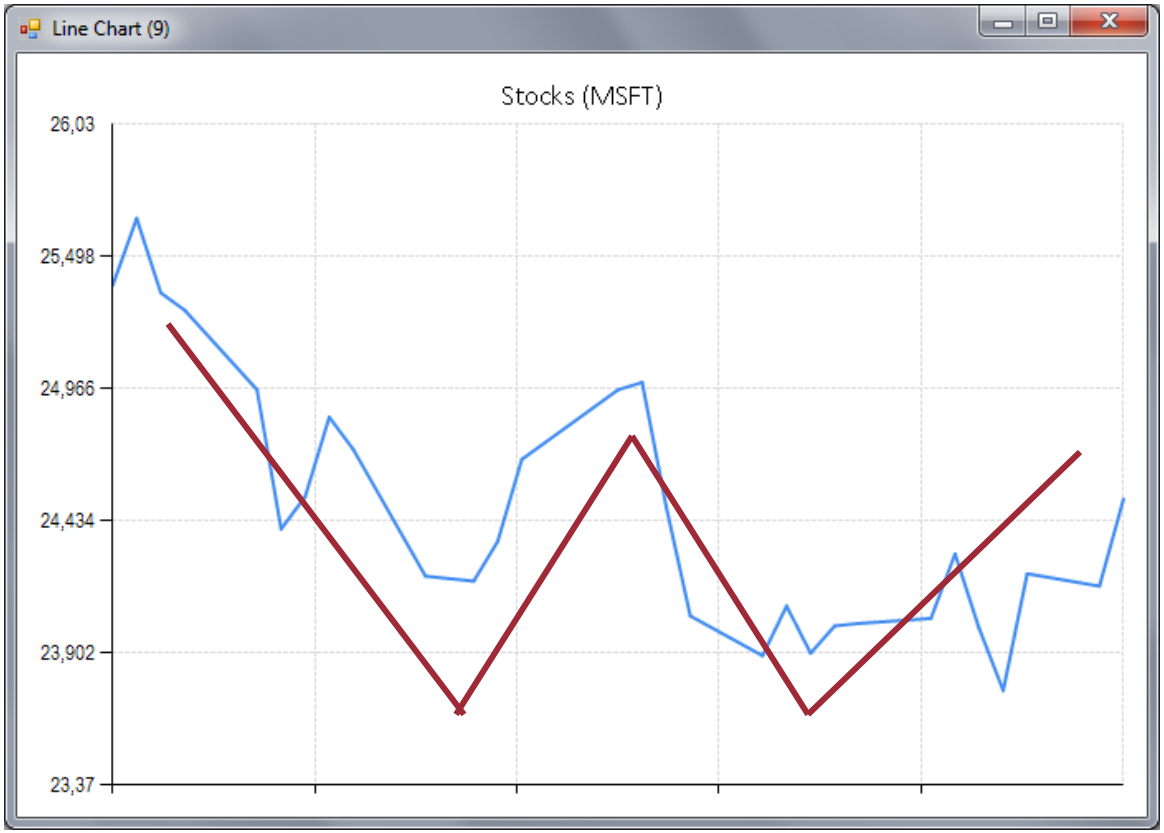
Declining pattern



Rounding top pattern



Multiple bottom pattern



Doman-specific language approach

Primitive classifiers

- **Declining** price
- **Rising** price

Combinators for classifiers

- Average using **regression**
- **Sequence** multiple patterns
- Check patterns in **parallel**

DEMO: CLASSIFIER DSL

http://fssnip.net/bK

DSL for price patterns

Conditions on **subsequent** parts

```
sequenceAnd rising declining
```

Run classifier over **linear regression**

```
regression declining
```

Run **two classifiers** and combine values

```
both minimum maximum |> map (fun (l, h) -> h - l)
```

Check that **both conditions** hold

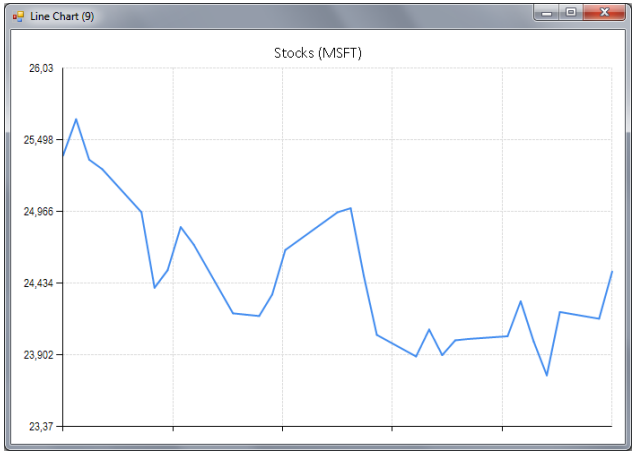
```
bothAnd declining (inRange 20.0 25.0)
```

Demos and Tasks

Double bottom pattern

Change over regression

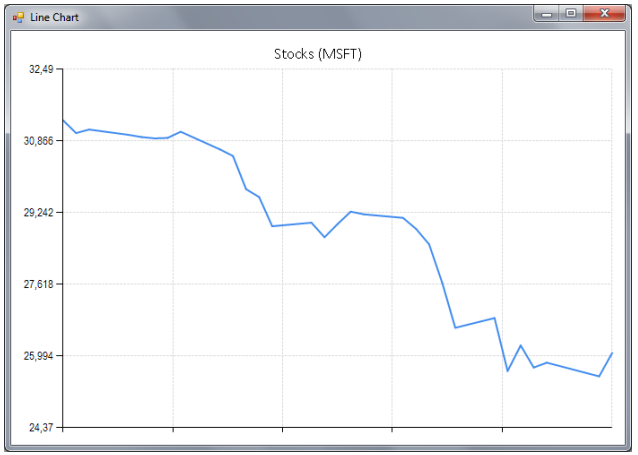
Down–Up two times



Declining fast pattern

Declining over regression

(Max – Min) > 3 USD



How does it **work**?

What is a **classifier**?

```
type Classifier<'T> =
  ClassifyFunc of ((DateTime * float)[] -> 'T)
```

A **function value**!

- Given data, calculate the result
- Generic – can produce anything
- Abstract – only internal

Complex from simple

Check **multiple** conditions

```
let bothAnd a b =
  both a b |> map (fun (a, b) -> a && b)
```

Calculate **minimum** value

```
let minimum =
  reduce min |> map (fun v -> Math.Round(v, 2))
```

All values are in a **range**

```
let inRange min max =
  bothAnd (atLeast min) (atMost max)
```

Domain-specific languages

Using **discriminated unions**

Using **function values**

- Defines the **representation**
- **Unlimited** expressivity
- **Run** the function to evaluate

Advanced **domain-specific language** topics

Types and DSLs

Grammar for the language

How to compose primitives?

```
average : Classifier<float>
```

```
lessThan : float -> Classifier<float> -> Classifier<bool>
```

Other benefits

Editor **IntelliSense**

Useful **documentation**

Repeating patterns

Repeating functions in DSLs

Map – transform the produced value

```
('T -> 'R) -> Classifier<'T> -> Classifier<'R>
```

Merge – combine results of two tasks

```
Classifier<'T1> -> Classifer<'T2> -> Classifer<'T1 * 'T2>
```

Simplify using them? With **language syntax**?

Computation expressions

Syntax for computations

For types with **certain operations**

Aka **monads** in Haskell

Declining without fluctuation

```
classify {
  let! lLo, rHi =
    sequence minimum maximum
  let! down = declining
  return down && (lLo >= rHi) }
```

DEMO: CLASSIFIER “MONAD”

Conclusions

Conclusions

Domain-specific languages

- **Choose** a **class** of problems
- **Design** simple **primitives**
- **Use** powerful **composition**

The **F# language**

- Discriminated unions
- Typed functional style