Investment Strategy Language (ISL)
DSL Report Card

Durga Harish Dayapule
School of EECS
Oregon State University

June 13, 2018

1 Introduction

The need to write flexible investment strategies is a primary job role for a financial analyst in finance industry. These Strategies are subjective decision made by the analyst in his/her daily job. The current project outcome is an investment Strategy language or (ISL) which gives the flexibility for a non-programmer to write strategies in an user friendly way. By defining patterns one can take a concrete action to buy/sell a stock. Chart patterns define the movement of raw price like closing or opening price or their derivatives. For example, moving average is a running average of price. We call these raw prices and their derivatives technical indicators, because they indicate the state of the market. Indicators like moving average are mathematical formulae expressed in terms of the prices/volume associated with a discrete set of time periods

2 Users

This DSL can be used by a non-programmer with a finance background. This DSL is motivated at different levels of users a) Financial Analysts b) Stock Broker c) Portfolio Manager d) Any other user interested in investing his/her money in Stock Exchange.

The User should have basic understanding patterns, so this can allow them to express complex patterns. Currently, the DSL is not at the user interaction stage. I have to work on thinking how a ideal user wants to interact with this DSL. The Current DSL only allow to take patterns and create investment strategies. But, In general it is not realistic. There can be many other parameters which affect an investment of money and time. The DSL has to expand its horizon to allow more Observations/parameters. For instance, Current financial analyst decide based on sentiment of the company, fundamental data of the company (Revenue, Profits), who other companies in the same sector are performing. With the current DSL, a user can not express any of these. As a user, he/she aims have a unified language were he/she can plug and play with these parameters an build their own strategies.

3 Outcomes

I have implemented the Investment Strategy language as two mini Deep DSL's. One which expresses patterns and other which changes the portfolio state (Cash,Stocks). The main outcomes of the DSL are, (i) Able

to express complex financial patterns. For example, Patterns such as Pennant, Cup And Handle, Head And Shoulders patterns which gives information to buy/ sell stocks when these pattern exists in the technical data of a company. (ii) Able to check these patterns on multiple companies stock data and then decide to take a sell or a buy action.

Another important outcome is to use this pattern matching DSL to action, by that I mean using these pattern matching operation on timeseries and making a real action to buy or sell. For doing this we use portfolio Deep DSL to express.

In order to see the output of the DSL, a person need to write a program defining user actions such as loading Cash, unloading existing Cash, Buying Stocks at time t, Selling Stocks at time t.

4 Use Cases / Scenarios

Example 1 : A simple Up Pattern

```
basicPat :: Pat
basicPat = Bas Up 2 4

--running program
matchPat basicPat openData

--output = False
```

Example 2 : for all Pattern

```
exPatFor :: Pat
exPatFor = (forAllPat (20,30) 1 Up)

--Running program
matchPat exPatFor  openData

--output = False
```

Example 3 : Complex Pattern

```
exPatComplex :: Pat
exPatComplex = And (patMin exPatFor) (patMin (shiftPattern (patMin exPatFor) 11))
```

Example 4 : Example using lift2Match

```
outPat :: [(TimeStamp,TimeStamp)]
outPat = lift2Match 15 (patNorm exPatComplex) openData 470

-- Exceution
outPat

--output
-- [(21,30),(41,50),(51,60),(71,80),(101,110),
--(121,130),(141,150),(151,160),(161,170),(181,190),(201,210),
--(211,220),(221,230),(251,260),(261,270),(271,280),(281,290),
--(291,300),(301,310),(321,330),(341,350),(351,360),(361,370),
--(371,380),(381,390),(401,410),(431,440),(441,450)]
```

Exploring Portfolio Deep DSL.

Example 5 : Example of loading Cash and unloading cash from a user

```
exProg1 :: [Prog]
exProg1 = [(L 1000 2),(UnL 100 3)]
```

Example 6 : A sequence of Actions

```
exProg2 :: [Prog]
exProg2=[
(L 1000 2),    -- Loading 1000 units of cash at time 2
(UnL 100 3),   -- Unloading or withdrawing 100 units of cash at time 3
(B 4 [(2,"CVS"),(1,"Apple")]),  -- Buying 2 stocks of "CVS" and 1 Stock of "Apple" at time 4
(IE ((matchP (shiftPattern (patNorm exPat3) 15)) "Apple"))  -- If the pattern matches
(B 25 [(2,"Apple")] ) -- Then Buy 2 "Apple" stocks at time 25
(B 40 [(2,"Apple")] ) -- Then Buy 2 "Apple" stocks at time 40
]

-- Execution of the above program.
listSemProg (USD,0,Zero) exProg2 -- the (USD,0,Zero) (Currency Type,
--Initial Cash of the user, Zero means there --are no stocks)
--output : (USD,405.57004,A (A (A Zero (Scale 2.0 (Stock 4 C1))) (Scale 1.0 (Stock 4 C23))) (Scale 2.
```

5 Basic Objects

As discussed previously, the complete DSL is depends on two Deep DSL a) Pattern DSL b) Portfolio DSL. In this section I will highlight the data types used in these DSL's. I will highlight the important objects used for both DSL's

TimeStamp : TimeStamp object defines the calender date, for simplicity purpose I have considered it as a Int Object. This can be extended to a real Timestamp object.

TimeDelta : TimeDelta object defines the difference of two TimeStamps objects, this object gives the flexibility to add or subtract times.

TimeSeries : A TimeSeries Object defines the mapping of time to prices/Volume. For this DSL, I have used it as a function from TimeStamp to type a.

Pattern Objects:

Base Patterns : A pattern is build from basic patterns (UP, Down, Cons, NoPat). =In this DSL, a pattern inherently has to be represented by start time and end time of that Pattern. For Example, when we define `Bas Up 1 6`, this means that the pattern is defined between TimeStamp 1 and TimeStamp 6. We can also extract the timestamps of the basic pattern and define the delta where the pattern is defined. This can be a future work for this DSL. The Base Pattern The Base pattern objects are Up which defines if the pattern is moving upwards. Down, if the pattern is moving Downwards. Cons, to represent a constant pattern across the given time.

And Pattern : The And Pattern Object allows to define combined patterns. The And P1 P2 means that both the patterns has to exist together.

Or Pattern : The Or pattern is used to mean if any of the pattern occurs in a Timeseries.

IfElse Pattern : The IfElse pattern allows a user to define patterns based on some condition.

```
data Dir = Up | Down | Cons | NoPat
        deriving(Show,Eq)

data Pat = Bas Dir TimeStamp TimeStamp
          | And Pat Pat
          | Or Pat Pat
          | IfElse Cond Pat Pat
```

Stocks DSL

Currency : This Object is used to define currency type for buying and selling stocks. These consists of USD, EURO. This Object allows an user to buy in one currency type and exchange in different currency.

Company : This Ojbect allows us to define Company Objects. In the DSL, the company names, company timeseries are mapped to the company object. This Company object allows us to interact with the DSL in a human understanding language.

Stocks : The Stocks Object defines types of stock a user holds. The Stocks Object has the information of when the stock was purchased, company name and TimeStamp when it was bought. This object allows us to express any kind of Stock. The Scale data type in the the Stocks Objects allow to scale any kind of stock. **Cash** : The Cash Object stores the information of the user cash. This is defined on a currency type.

```
data Cur  = USD | INR | EURO

data Comp = C1 | C2

data Stocks = Zero
              | Stock TimeStamp Comp
              | Scale Float Stocks
              | A Stocks Stocks

type Obs a = TimeStamp -> a

data Cash      = C Cur Price
```

Program Object : This Object allows users to interact with the Investment Strategy language in a human understandable way. The list of Prog defines the sequence of operations to be done on Stocks object. The ”IE” data constructor allows the user to define patterns in the TimeSeries using the Bool data type.

```
type StockProg = [Prog]

data Prog = L Float TimeStamp
          | UnL Float TimeStamp
          | B TimeStamp [(Float,String)]
          | S TimeStamp [(Float,String)]
          | IE Bool Prog Prog
```

The Current limitation, these object inherently binds the semantics. This kind of definition makes the current version of DSL to be Partially Shallow DSL and more like a Deep DSL. In the future work, I would like to open up these objects and make the current DSL to a complete Shallow DSL.

6 Operators and Combinators

The Pattern DSL has the following Combinators to define more complex patterns.

```
patPlus ::  Pat -> Pat -> Pat
```

This combinator allows to define addition on two patterns. A addition on two patterns is an and operation of the two patterns.

```
patMin ::  Pat -> Pat
```

This operator is used to negate a pattern. The negation is defined on basic patterns. The negation of Up is Down, Down is Up, Cons is Cons, NoPat is NoPat. This negation operation allows the user to flip the pattern easily.

```
shiftPattern ::  Pat -> TimeDelta-> Pat
```

A Pattern in the current DSL has start timestamp and end timestamp of the pattern. The shiftPattern operator takes the current Pattern and shifts by timedelta t, the shift operation changes the start time and end time of the pattern.

```
patNorm ::  Pat -> Pat
```

In the current DSL, the pattern has starttime and endtime information. So, we define a patNorm operator which normalizes the pattern to base timestamp (starting timestamp of the time series). This will allows the user to compose different patterns in a easy way. We will use illustrations to show the power of this combinator.

```
patEquals ::  Pat -> Pat -> Bool
```

This combinator allows the user to check if a pattern p1 equals to p2.

```
forAllPat ::  (TimeStamp,TimeStamp) -> TimeDelta -> Dir -> Pat
```

The forall combinator allows to construct a composition of And patterns over a defined time interval (t1,t2) using basic directions. Its a function which takes a tuple (t1,t2) and timedelta which defines the window to move from start time t1 to reach end time t2, and basic direction (Up,Down,Cons) and produces a pattern.

```
applyTimeToBasic ::  [(TimeStamp,TimeStamp)] -> [Dir] -> [Pat]
```

applyTimeToBasic operator takes a list of start and end timestamps, and a list of directions and produces list of patterns.

```
matchPat ::  Pat -> TimeSeries a -> Bool
```

This operator is used for checking if a pattern exists in the Timeseries a . It takes a pattern and timeseries and return boolean (true/false)

Examples based on above fundamental patterns

```
p1 :: Pat
p1 = (forAllPat (320,350) 5 (Down))
```

The above forAllPat defines between start time 320 and end time 350 with timedelta of 5 and basic pattern Down. This means that each point between 320 and 350 with timeinterval 5 are below the value of the timeseries at 320(start time).

```
listPat1 ::[Pat]
listPat1 = applyTimeToBasic [(1,2),(2,3),(5,6)] [Up,Up,Down]
```

This example outs a list of patterns.

```
exPatComplex :: Pat
exPatComplex = And (patMin exPatFor) (patMin (shiftPattern (patNorm (patMin p1)) 11))
```

This example has more complex pattern compositions. The patMin operation flips the pattern p1, the next operation is patNorm which normalizes the pattern (i.e) it changes the pattern start and end time, next the shiftPattern shifts the pattern by timedelta of 11 (this operation changes the pattern starttime and endtime). Again we do a negation of the whole pattern. At last, we combine two pattern by using And data type. This example gives a flavor of how to define a complex pattern with these operators.

Higher Order Functions

```
lift2Match ::  TimeDelta -> Pat -> TimeSeries a -> TimeStamp ->
[(TimeStamp,TimeStamp)]
```

This lif2Match function takes timedelta, a pattern, timeseries, end timestamp and outputs list of matching time intervals. This function is important when the user want to query on the the time series.

Example of lift2Match

```
outPat :: [(TimeStamp,TimeStamp)]
outPat = lift2Match 15 (patNorm exPatComplex) openData 470
```

The above example takes the complex pattern (exPatComplex) and normalize it. Once the pattern is normalized then we can use the lift2Match function and check the pattern exists by moving the pattern by 15 units in time till we hit endtime of 470.

```
matchAllC ::  Pat -> [String] -> Bool
```

This function takes a pattern and takes list of company names and checks if the pattern matches in all the companies.

Portfolio Operators

```
buyStock ::  (Cash,Stocks) -> Action -> (Cash,Stocks)
```

This operator allows to change the current (Cash,Stocks) state by taking a Buy or a Sell Action.

```
sellStock ::  (Cash,Stocks) -> Action -> (Cash,Stocks)
```

The sellStock function changes the current (Cash,Stocks) state by taking a Buy/Sell Action. The Action contains the information of which stock to sell at time 't'. But, In general the Stocks data consists of different shares bought at different times. So, the sellStock method uses max profit at time t and sell place as a queue and returns an updated queue.

```
listSemProg ::  (Cur,Price,Stocks) -> [Prog] -> (Cur,Price,Stocks)
```

This function executes the [Prog] (i.e) Sequence of actions (Load, Unload, Buy, Sell). This takes the current state of the (Cash,Stocks) and takes a [prog] and generates the next state (Cur,Price,Stocks). Please note there is a mapping function which converts (Cur,Price) to Cash data type. This flexibility is given so that user can define in a natural way.

7 Implementation Strategy

The current implementation of the DSL can be classified towards Deep DSL. In the Pattern DSL, there are fewer compositions that can be done on base pattern. For example, two patterns can be composed using an "And" operation or "Or" Operation. This motivated me to include these operation in abstract data type. Such a definition has disadvantage not to exploit the functional way of doing things. As the definition and properties of a pattern was clear, this allowed me to express a pattern algebra for the Pattern DSL. In the

pattern DSL, The higher order functions like (lift2Match) uses these lower level functions (matchPat) using map operation.

In order to fully exploit the features of both these Deep DSL's. The Investment Strategy language provides another Deeply embedded DSL which can be used to execute strategies. This Deep DSL uses the operators from Pattern DSL and portfolio DSL.

8 Related DSLs

The paper on composing contracts[1] is one of the most relevant paper I refereed while working on my DSL. The composing contract paper provides a clean implementation of scaling contracts and defining currency as different data type. This was used in the current implementation for scaling Stocks. This paper is more topically related, this paper provided a clear understanding of what types should be and how they can be used.

I also found Charting Patterns[2] on Price History paper related to my pattern DSL. They also similarly try to match the patterns for timeseries data.

9 Future Work

The current DSL is limited for a real user to use. As mentioned in Users sections, in general a financial analyst has to consider different observations to come up with as strategy. So, in future, I would like to work on implementing any kind of observations to the data model.

The Syntax of the current DSL is not rich enough for a user to write programs. I would like to implement a parser for defining these strategies and execute the program. For example, the current DSL can not handle this situation : For all the companies choose the companies having pattern "a" and buy stocks which satisfy the matching condition. I need to express first order logic operations in future, this can really help a user to express and think of strategies.

10 Running the Haskell Code

Dependencies : These are dependencies need to run and use the Haskell Code.
Prelude
System.IO
Data.Function (on)
Data.List (sortBy)
haskell_chart library

Module Name and Description:
1. BasicTypes : This contains all the ADT's for the DSL language.
2. CompanyCash : This contains 120 company time series data, their mapping to Company object, currency, company name.
3. ISLExamples : This has few examples (or) helper examples.
4. PatternAlgebra : This module has all the pattern algebra operations, compositions of the patterns.
5. PatternHelpers : This module is a helper module for PatternAlgebra 6. PortfolioAlgebra : This module has portfolio Operations. 7. TimeMethods : This has methods related to TimeStamp, TimeDelta etc. 8.

TimeSeriesHelpers : This module is used for plotting the timeseries and patterns.

Running a program:

open ghci and load following modules and use examples in the section 4 to see the output.
:l PortfolioAlgebra.hs CompanyCash PlottingData.hs

References

1. Composing Contracts: An Adventure in Financial Engineering, Simon Peyton Jones
2. Charting Patterns on Price History Saswat Anand, Wei-Ngan Chin, Siau-Cheng Khoo ACM SIGPLAN International Conference in Functional Programming (ICFP), 2001