LAZY-STREAMS APPLIED TO FOREIGN EXCHANGE ARBITRAGE

CONNIE LIU & JAMES UTTARO

CSc 335: PROGRAMMING LANGUAGE PARADIGMS

TABLE OF CONTENTS

OBJECTIVE..... 3

IMPORTANCE OF STREAMS IN FUNCTIONAL PROGRAMMING..... 4

DIFFERENCE BETWEEN LAZY AND EAGER EVALUATION 4

LAZY STREAMS CONNECTION TO FOREX ARBITRAGE.....5

OUR MODEL FOR IMPLEMENTING FOREX ARBITRAGE USING STREAM 9

BELLMAN-FORD ALGORTIHM 13

CONCLUSION.....15

REFERENCES.....14

OBJECTIVE:

Streams provide a way of modeling state without using assignment; their implementation via delayed evaluation permits the representation of very large (even infinite) sequences. Our objective is to use streams to represent our graphs queried from real time foreign exchange market rates. Using streams we can handle real time data in time intervals instead of evaluating a batch data set all at once. Throughout the day, data sets are continuously being queried in and the streams will be very large. With lazy-streams we can evaluate each object at a time and check for arbitrage only when needed. This will avoid time complexity issues and other complications such that the market rates are constantly being updated.

Importance of Streams in Functional Programming:

Streams are like iterators, but with list syntax. They work using lazy-computation which means the next element is not evaluated until it is needed. Once a stream calculates its rest, it won't recalculate it again; it will have the evaluation value stored. The technique of delayed evaluation enables us to represent very large (even infinite) sequences as streams. Stream processing lets us model systems that have state without ever using assignment or mutable data. Streams allow one to use sequence manipulations without the cost inefficiencies

of manipulating sequences as lists. With streams can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation. The basic idea is to construct a stream only partially, and to pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough of itself to produce the required part, thus preserving the illusion that the entire stream exists.

In general, delay is crucial for using streams to model signal-processing systems that contain loops. Without delay, our models would have to be formulated so that the inputs to any signal-processing component would be fully evaluated before the output could be produced.

According to the article " The 8 Requirements of Real-Time Stream Processing" applications that require real time processing of high volume data streams are pushing the limits of traditional data processing infrastructures. Using real time data from a conventional database is not always reliable. In a real-time system, since the data is never stored, data is constantly late, delayed, missing or out of sequence. The author claims that there needs to be requirement during this step to individually single out the evaluations and "block computations" while waiting for the input instead of continuing to run the program. There must be a "time-out" which lazy-streams can come into action. Using lazy streams we can singly handle single elements in our stream and have a break with a promise to evaluate the next waiting input in a timely fashion. Streams also provide a way to

essentially store data that has been evaluated so it does not need to be evaluated again.

Difference Between Eager and Lazy Evaluation:

The difference between eager and lazy evaluation is “call by value” and “call by need.” Eager evaluation evaluates the argument before the performing the function application. However, lazy evaluation or “call by need” will only evaluate an expression when the value is needed. In a lazy language, f(exp) = 1, even if exp fails to terminate — the argument to f doesn’t need to be evaluated to determine the result. This isn’t done by analyzing the definition of f, but rather by delaying evaluation of the argument until its value is required. Not evaluating an unused argument saves time. If its evaluation does not terminate, not evaluating it saves a lot of time!

Lazy-Streams connection to Foreign Exchange Arbitrage:

Foreign Exchange Arbitrage exploits a price difference between two or more markets with the ability to make money with zero risk. A profitable trade is only possible if there exists market imperfections. When one market is undervalued and one overvalued the arbitrageur creates a system of trades that will force a profit out of anomaly. We will use lazy evaluation to check for anomalies and force trade using lazy evaluation only when the discrepancies exist.

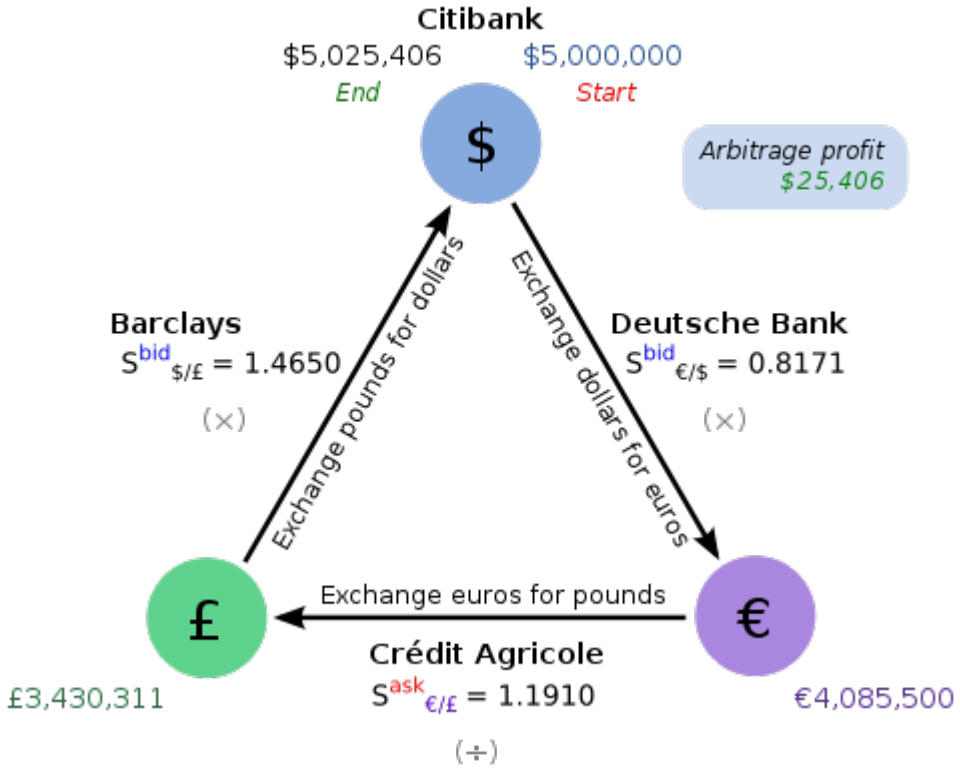

Figure 1: Forex Triangular Arbitrage

Figure 1 above shows an example of an arbitrage opportunity. Where the discrepancies in currency exchange between banks have the potential to make a profit.

Throughout the trading hours of the day currencies are consistently traded, likewise the value of each currency is updated following information gathered throughout 10-minute intervals. This results in opportunities where the exchange rates between currencies are not exact. In turn we can find situations where there is a potential fortune to be made in currency trading by analyzing real-time currency exchange rates to find a way to turn $1 USD into more than $1 by a sequence of trades. Consider the example above in Figure 1. Starting with $5

million USD and exchanging to EUR at a rate of .8171 EUR/USD we end up with € 4.0855 million, this can then be traded into GBP at a rate of 1.1910 EUR/GBP giving us £3,430,311, after this exchange if the discrepancy we have found is profitable we will find in the next exchange a profit when we exchange back to the starting currency, USD. Here we see that when we exchange £3,430,311 to USD at a rate of 1.4650 USD/GBP we end up with $5,025,406. Therefore netting an arbitrage profit of $25,406

Lazy-streams can be used to force trades only when a discrepancy occurs. It is more efficient to only force trade when a profitable discrepancy exists else, it would be very time consuming to continuously evaluate the functions and check for anomalies. Lazy-streams would allow us to delay evaluation of the argument until its value is required. Not evaluating an unused argument saves time. If its evaluation does not terminate, not evaluating it saves a lot of time! The initial exchange rate is usually most of the time static; therefore the trade using lazy evaluation will only be initiated when the rate changes to a profitable number. There is also huge amounts of data and rates consistently coming in every minute therefore it would be very time consuming to continuously check and trade and hope for a profitable increase in foreign exchange currency rates.

OUR MODEL FOR IMPLEMENTING FOREX ARBITRAGE USING STREAMS:

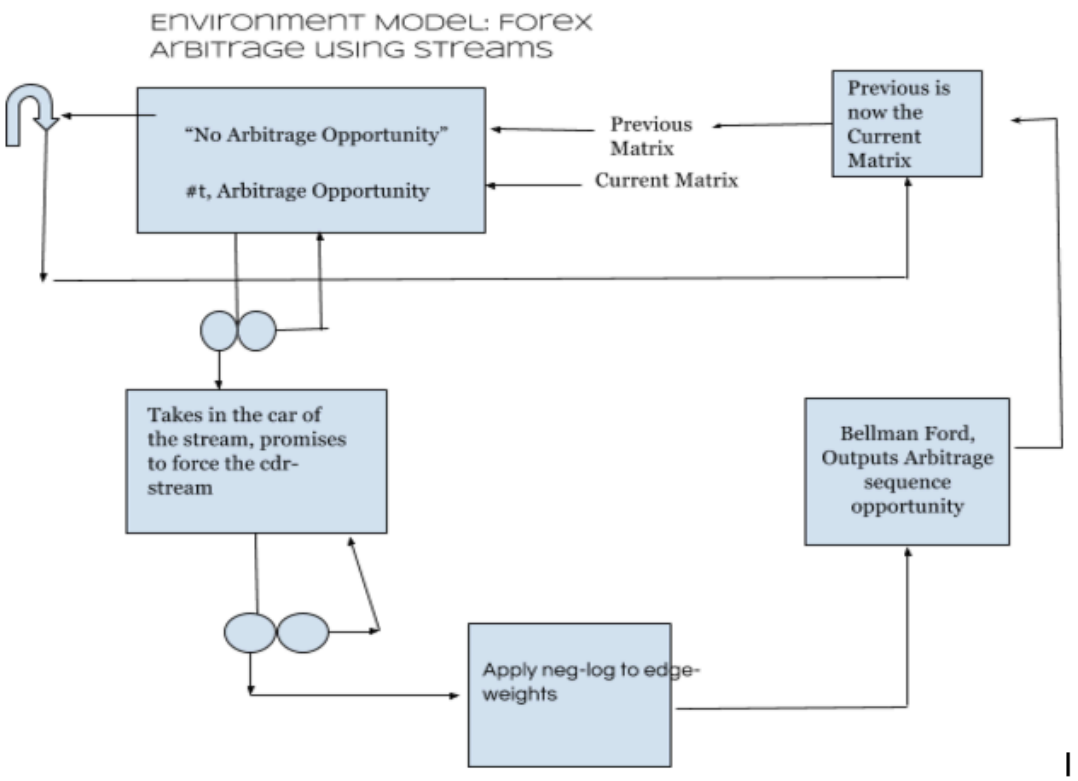

Figure 1: Environment Model for Forex Arbitrage using Streams

Our data type will be a directed weight graph represented as two lists with the first list as our vertices (currencies) and our second lists as our edges (exchange rates). Our project will output the sequence of currencies with a profitable arbitrage (USD » EUR » CAD) and output the value of profit that can be achieved from each currency (+ 1.008).

We will implement foreign exchange arbitrage using streams and applying the Bellman Ford algorithm which will find the shortest path of the matrix, (which can also be represented as a graph) and this will return the sequence of

currencies needed to perform the arbitrage and the profit opportunity that can be achieved.

Rates	USD	EUR	GBP	CHF	GBP
USD	1	.741	.657	1.061	1.005
EUR	1.349	1	.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	.942	.698	.619	1	.953
CAD	.995	.732	.650	1.049	1

Figure 2: Currency Exchange Rates

Summarizing our model, we express the arbitrage problem in terms of a graph with weights on the edges. A vertex in the graph represents each currency. Since we can convert from any currency to any other, our graph contains all possible edges between two currencies. We want to find a profitable conversion sequence by somehow assigning weights to the edges so that a profitable conversion sequence shows up as a negative weight cycle. The length of a shortest path in a graph is expressed as the sum of the edge weights in the path, while the rate of profit on a cycle in our graph is expressed as a product of conversion rates. But we know that $M[usd, eur]*M[eur,gbp] \cdots *M[ik-1, ik] > 1$ if and only if the $\lg M[usd, eur] + \lg M[eur, gbp] + \ldots + \lg M[ik-1, ik] > \lg 1 = 0$ In other words, if we assign a weight of $\lg M[i, j]$ to each edge, a profitable cycle is a positive-weight cycle. If instead we assign a weight $w(i, j) = -\lg M[i, j]$ to each edge (i, j), a profitable cycle appears as a negative weight cycle in the graph. To find out if the

graph contains a negative-weight cycle, we run the Bellman-Ford algorithm, and then test each edge (i, j) to see if d(j) > d(i) + w(i, j). If this is true for any edge, then we know the graph contains a negative-weight cycle.

To actually print out the sequence of vertices in some negative-weight cycle, we need to store extra information with the shortest-paths. We keep an array of predecessor pointers p(i), along with the path-weight matrix d(i). Every p(i) is initialized to NULL. When running the Bellman-Ford algorithm, each time we reduce the path length to vertex vj by setting d(j) ← d(i) + w(i, j), we set p(j) ← i, to indicate that the shortest path found so far to vj ends with edge (i, j). The predecessor pointers define a graph on the vertices. Claim: After we have run the Bellman-Ford algorithm, any cycle in the graph defined by the predecessor pointers is a negative-weight cycle.

Proof: Consider the last edge (i, j) in the cycle for which p(j) was set by RELAXing edge (i, j). Let (j, k) be the next edge in the cycle. Since the value d(j) was reduced since d(k) was set, we could reduce d(k)by setting it equal to d(j) + w(j, k). Similarly we can reduce the d() value for every vertex in the cycle, finally reducing d(i) and demonstrating that this is a negative-weight cycle. To find a cycle in the graph of predecessor pointers, we can use depth-first search.

The code for our stream-based function is shown below:

```
; We are using streams to handle real time currency exchange rates different intervals of time during the day
; instead of evaluating a batch data set all at once. It could be 10 second intervals or in one minute intervals, either way
; we will have large data which we represent as graphs coming in.
; For simplicity reasons we hard coded graphs/matrices based upon real time foreign exchange market rates.
; Instead of creating a program that would take data from online and important it into our functoon
; Eventually streams would allow us to have infinite data sets without time complexity issues and other complications
; Graph-stream is the stream we have manually coded based on real foreign currency exchange rates
; (best case scenario we will query data from an online exchange rates server but for now this is to show our objective)
; we have three graphs that we would like to find a profitable arbitrage if any.

(define graph-stream1 (cons-stream A (cons-stream B (cons-stream C null)))) ;goes to bellmanford
(define graph-stream2 (cons-stream B (cons-stream A (cons-stream C null)))) ;no arbitrage eventually returns empty list
(define graph-stream3 (cons-stream A (cons-stream A (cons-stream A null))))
;stream-arb takes the stream, previous graph or original and the type of currency we are starting with
(define (stream-arb stream-dw-graph prev-graph source)
  (let (( cur-graph (directed-graph (get-edges (stream-car graph-stream1)) (neg-log-weights (stream-car graph-stream1)))))
  ;(let ((cur-graph (directed-graph (get-edges (stream-car stream-dw-graph)) (neg-log-weights (stream-car stream-dw-graph)))))
    (cond ((stream-null? stream-dw-graph) the-empty-stream)
        (else (cond ((eq? (check-graphs (car stream-dw-graph) prev-graph) #t)
                     (stream-arb (stream-cdr stream-dw-graph) (stream-car stream-dw-graph) source)) ;no arbitrage catch
         ;(cond ((equal? (stream-car stream-dw-graph) prev-graph)
                    (stream-arb (stream-cdr stream-dw-graph) (stream-car stream-dw-graph) source)) ;; no arbitrage catch statement
                    ((eq? (check-graphs (stream-car stream-dw-graph) prev-graph) #f) (my-bellman-ford cur-graph source))
                    (else ((stream-arb (stream-cdr stream-dw-graph) (stream-car stream-dw-graph) source)))))))

(stream-arb graph-stream1 C 'eur); bellman-ford: negative weight cycle
(stream-arb graph-stream2 A 'usd) ; '() no arbitrage found
(stream-arb graph-stream3 A 'eur); bellman-ford: negative weight cycle
```

Figure 3: Stream based Arbitrage code

Bellman-Ford With Respect to Currency Exchange Rates:

Bellman-Ford Algorithm is a single-source shortest path class procedure, which finds the shortest path from a single source vertex to all vertices in an edge-weighted digraph. Similar to Dijkstra's algorithm in terms of its procedure class, it differs from Dijkstra's with its ability to deal with negative cycle. Bellman-Ford follows a similar relaxation pattern of edges like Dijkstra's; it gradually finds the shortest path from the source vertex to all vertices in the graph and updates the path upon each iteration. Both algorithms use an overestimation of the length of edges through each passage across the graph to

find the optimal solution. Unlike Dijkstra's greedy queue approach, bellman-ford initializes all edges to be relaxed. This allows Bellman-ford to be as accurate as possible at every given iteration, this approach allows Bellman-Ford to be applied in a wider variety of applications. Bellman-Ford's negative cycle handling is highly beneficial in the context of Forex Arbitrage. We will consider Currency Exchange Table 1, which shows the conversion rates among currencies with five vertices. We define an arbitrage opportunity as a directed path (cycle) such that the exchange of currencies by a sequence of trades is greater than one. In example we see (USD >> EUR >> CAD) to be .741 * 1.366 * .995 = 1.008.

Our approach to apply Bellman-Ford with respect to Forex Arbitrage as a negative-cycle detection problem, we will want to transform the weight of each vertex with the logarithm of the weight negated (-log (weight-of-vertex)). With respect to the original problem of multiplying each edge-weight of a given sequences, we approach the problem as computing the path-weight as adding the edge-weights together giving us largest negative path. This will translate into the most profitable sequence of trades, hence finding the shortest path from the source node.

LIMITS OF ARBITRAGE:

According to "The Limits of Arbitrage" due to restrictions placed on funds that would normally be used by traders to arbitrage away pricing discrepancies, prices may remain in a non-equilibrium state for expected periods of time. The efficient market hypothesis (emh) assumes that when misplacing occurs on

publicly traded stock, there is a opportunity for low-risk profit. The low-risk profits exist as a side effect of triangulation arbitrage. If the stock falls below equilibrium price due to irrational trading, rational investors take a long-term position. Rational traders tend to work with professional investment firms. If these firms engage in arbitrage in reaction to a stock misplacing, if the misplacing lasts for an extended period, clients of a firm tend formulate the opinion that the firm is incompetent. Intern clients tend to withdraw investments, the threat of this causes firms to be less vigilant of arbitrage opportunities, this has a tendency to worsen the problem of pricing discrepancies.

CONCLUSION:

Streams provide a timeless approach to lists, in which there implementation delay an object, this object will eventually be forced to evaluate. In turn this allows one to deal with very large or pseudo-infinite sequences. Potential future implementation would be able to handle querying live data from a foreign exchange rate aggregation server, which would be complementary to the property of streams delayed and forced objects. This would allow us to deal with time intervals at a single intense, facilitating fast arbitrage predictions that can be passed onto a trading scheme. Streams implicit lazy-list implementation let us evaluate each object one at a time and check for arbitrage only when needed. This will avoid time complexity issues and other complications such that the market rates are constantly being updated.

REFERENCES:

Chang, Stephen. "Racket Generic Graph Library." *Racket Generic Graph Library*. N.p., n.d. Web. 27 May 2016.

Stonebraker, Michael, Uğur Çetintemel, and Stan Zdonik. "Real-Time Transaction Processing." *SpringerReference* (n.d.): n. pag. *The 8 Requirements of Real-Time Stream Processing*. Web. 27 May 2016.

Andrei Shleifer and Robert W. Vishny, 1997, 'The Limits of Arbitrage', The Journal of Finance, American Finance Association