UCL

MSC Computer Science

Project Report

Modeling The Financial Markets

Author:
Peter Hilton

Supervisor:
Dr. Christopher D. Clack

November 1, 2012

Abstract

This project is about the modeling of financial markets. Several approaches to this modeling task are assessed, against an example hypothesis — an example of the sort of problem we want to be able to be able to easily model. Three modeling formalism are assessed: the pi calculus, a mathematical approach and the Stochastic Pi Calculus. The information gained from this assessment is then used to design an improved formalism for this modeling. Finally this new system is given a partial implementation.

Contents

Chapter 1

Introduction

1.1 Motivation

Much of modern economic theory is based upon the idea that the market price of an asset accurately reflects the underlying value of an asset. A markets assure the efficient allocation of capital, meaning capital is transferred to where it is most useful. Fast-growing sectors of an economy have greater need for capital and the markets meet this need leading to greater capitalization of growing sectors which produces growth. As such, Dr. Willem F. Duisenberg, the first president of the European Central Bank claimed that "the best contribution that monetary policy can make to the smooth functioning and integration of European financial markets and to economic growth is to maintain a steady medium-term (market) price stability orientation"[11].

However, the markets have change significantly in recent years. The introduction of algorithmic trading has transformed markets and created new sources of instability [10]. These algorithms can interact in vicious ways, in-

troducing phenomena found in concurrent computing such as feedback loops, and this increases the instability of the markets. Thus this project is aimed at applying types of analysis commonly used in concurrent computing to the financial markets. Simply put, markets behave as distributed computer systems so they should be examined as such.

The most dramatic illustration of this new instability is the 'Flash-Crash' which hit the US equity markets on May 6 2010 (see section 2.2). During this crisis, several examples of instability were witnessed, but perhaps the most striking was the 'Hot-Potato' effect, where traders repeatedly bought and sold from each other with little overall change in net position (see section 2.2.1). During the course of this project, this effect will be the archetypal example of what we want to model. Existing modeling formalisms are assessed for the ease with which they can capture the essential behaviour of this system.

In order to model the system, a hypothesis of why it occurred would be needed. Previous research into the flash-crash has shown that delays in processing in the system created a feedback loop[17]. Delays in the system mean that algorithmic traders make their decisions based upon out-of-date information leading to traders making fundamental mistakes, such as underestimating their positions. It might be hypothesized that this is also the cause of the 'Hot-Potato' effect.

It is important to say that the aim of this project is not the exploration of this (or any other) hypothesis. Rather the aim is to explore the modeling formalism which supports such research and as appropriate contribute an extended formalism. The role delays in the Hot-Potato effect is an archetypal example of what we want to be able to model and is used to guide our

assessment of existing modeling formalisms.

1.2 Objectives

There are two objectives of this project. Firstly, existing ways of describing feedback loops such as the Hot-Potato effect have to be examined and their limitations found. Secondly, a new modeling formalism will be created which overcomes the problems of existing systems. This will then be suitable for use as a hypothesis formulation and exploration tool for future research into instability in markets as a product of interacting algorithms.

1.3 Contributions

During the course of this project, I make the following contributions.

- A description of the limitations of both a Differential-Mathematics and a π-calculus approach to modeling the interacting algorithmic traders in a market
- An encoding of several features of the functional-programming paradigm in the π-calculus
- The syntax and reduction semantics of an extended form of the π-calculus, where processing delays are taken as atomic actions
- The syntax and translation semantics of a high-level language, which marries functional-style syntax with the extended π-calculus

- A compiler and type-checker, which translates programs written in the higher-language to the extended π-calculus

In addition, the project includes some minor contributions:

- An encoding of integers in the π-calculus, based on church numerals, together with an abstraction which encodes the *plus* operation. This can for the basis of a complete definition of integer arithmetic in the π-calculus (Appendix C)

- A method of filtering a queue or other data structure in the pure π-calculus where the domain of member of the data-structure is infinite (Appendix E)

- Definitions in the π-calculus for the combinators S, I and K. As these combinators together define a system which is computionally complete[1], this could form the basis of a proof that the π-calculus iself is computationally complete (Appendix H).

- A discussion of strategies for turning this new static calculus into a executable modeling language (Appendix G)

Chapter 2

Background

2.1 The structure of Equity Markets

This project is interested in the US equity markets, and therefore it is their structure which is described here.

Equity markets are financial services which facility the exchange of equities — stocks and shares. Actors in the markets issue *orders* to an *exchange*. The exchange's responsibility is to accept these orders, and match trades where possible. When a trade occurs, the exchange notifies the traders involved. Modern exchanges are based upon the idea of an order-book.

2.1.1 The order-book

An order-book is the financial instrument through which trades are made. They work by traders submitting *orders* to the book. There are two main types of order, *limit* orders and *market* orders. Limit orders are offers to either buy or sell at a specific price. The pricing of limit orders is not con-

tinuous, but organised into ticks — thus each limit order must have a price corresponding to a specific tick or it is rejected. There are two good resons for this. Firstly, it greatly reduces the decisions involved with the pricing of orders, saving time in a fast-moving market. Secondly, it enforces time priorty in an order book, providing an incentive for a trader to provide liquidity[3]. The simplest order books have a constant difference between sequential ticks across the entire book, however this is not always the case — some books have a price difference between sequential ticks that varies with price.

Limit orders are promises to either buy or sell at a specific price. Thus if there is a buy and a sell Limit order at the same price, they will be matched, the traders informed of the trade, and the limit orders involved in the trade are removed from the book. This is one mechinism for trades to occur but is very rare. Usually there is a gap between the highest bid to buy an order, the *best bid*, and the lowest offer to sell, the *best offer*. This gap is called the *spread*. The midpoint between the best bid and best offer is called the market midpoint. If more than one limit order is sent at a particular tick price, the orders queue up in the order of arrival. Limit orders can be cancelled and removed from the book without a trade by sending a cancellation message to the exchange.

The second important type of order is the *market* orders. These are orders which are executed immediately (provided there are limit orders on the book), but with no guarantee on the price they will be executed at — they represend "'demand for immediate execution"' while a limit order represents "'supply of immediacy to other traders"',[12]. If there is a market order to sell, this will be matched will orders at the best bid price. These will

be matched from the front of the queue — i.e. the first limit order which arrived at that price will execute first. If a market order is bigger than the limit order it is matched with, the entire limit order will be matched and the remnants of the market order will be matched with the next limit order at the best bid price (this can be at a different tick). If the market order is smaller than the limit order which it is match with, the limit order will be partially filled, the unmatched part of it will remain on the order book.

2.1.2 Types of Trader

For the purposes of this project, I distinguish types the following types of trader:

- Fundamental buyers only buy stock. They can do this by either limit or market orders.
- Fundamental sellers only sell stock — again by either market or limit orders.

In addition there is a type of trader known as a *High-Frequency Trader*, or HFT, but as this shall be the focus of this paper, I will consider it in the next section. These are all the types of trader I will talk about. It is unsophistocated, but adequate for the purposes of this paper.

2.1.3 High-Frequency Trading

High frequency trading is a type of trading where many small orders are placed at very high frequency. Due to the speed at which occurs, it is entirely controlled by algorithms running on specially designed hardware. The

response time of such traders is measured in microseconds[5], and some HFTs are capable of executing millions of trades per day[9]. Such traders aim to make profit through their ability to react to market events quicker than other traders. They typically make a small profit per trade, but execute a high number of trades. I will examine two main strategies for HFTs: statistical arbitrage and market-making.

Statistical Arbitrage

Traders try to exploit statistical correlations between different securities to make profits. If an imbalance opens up, the HFT will quickly act to exploit the imbalance — predicting the price movement of the asset based on the price movement of the correlated asset. This is an important type of HFT algorithm, but it is not what we will be considering in this paper. HFTs mentioned in the subsequent sections will all be HFTs using the market-maker strategy.

Market-Making

The market-maker strategy aims to profit from the difference between the best-bid and best-offer for a specific equity. Given that the best bid is lower that the best ask, market-maker HFTs aim to buy at the lower price and sell at the higher price. Such trades use only limit orders (as market orders get the more unfavorable price). As they make trades, the inventory held by the HFT changes. HFTs may operate both *long* and *short*, meaning that the trader can borrow shares to sell (called selling short) as well as selling shares owned by the trader (selling long). A market-maker HFT aims to end

the day with a zero inventory, but may accumulate long or short positions during the day[2].

As mentioned above, market-maker HFTs aim to profit across the bid-ask spread. However, if the price of the asset is moving, they can lose money if the bid price when they bought is higher than the offer price when they sell. Therefore, the inventory held by a high-frequency trader presents a significant source of risk and HFTs act to keep this to a minimum.

2.2 Description of the Flash Crash

The 'Flash Crash' describes a short period of extreme volatility in the Chicargo Mercantile Exchange (CME). This in turn, caused sharp drops and subsequent recoveries in the American markets, with the Dow Jones Industrial Avarage (DJIA), NASDAQ 100 and S&P 500 all reaching daily minima between 14:45 and 14:47 ET (American Eastern Time.

The morning of the May 6 2010 started with falling markets. The Greek soverign debt crisis created increasing uncertainty in the markets – leading to an unremarkable increase in volatility in the markets[1]. At around 2:30 ET, this decline spiked and then recovered a matter of minutes later. Between 14:32 and 14:54:27, there was a 5.1% drop in the front-month (the next paying month) E-mini S&P 500 futures contract. At the bottom of the decline, another trade would have caused a price drop of 6.5 index points, or 26 ticks. This triggered the CMI Stop Logic. This pauses all trading on the order-book for five seconds if a trade would be executed more than six index

[1]http://www.sec.gov/sec-cftc-prelimreport.pdf

points away from the current price. During this period, limit orders can be submitted and cancelled on the book, but no trades are matched. At the end of this period, there was a brief period of fluctating prices followed by a rapid assent, as new buyers were attracted into the market[2]. This assent continued, occassionally interrupted, until 15:03 ET, when the recoverery levelled off at slightly below the price before the crash.

This behavour, although the most dramatic example, is not unique. In fact, there have been thousends of similar rapid declines and recoveries in markets since the introduction of High Frequency Trading [17].

After the flash crash, the Securities Exchange Commission (SEC) produced a report detailing what went wrong. It blamed a sell-off algorithm, selling USD 4.1 billion e-mini contracts - 150% of buy-side liquidity[9]. This algorithm was programmed to issue orders totalling nine percent of the total trade volume, calculated over the previous minute. Crucially, according to the SEC, this algorithm paid no attention to price or time. Previous trades of similar size and method took 5 hours to complete but, in the already stressed markets, this trade completed in just twenty minutes. This is because the increased trade volume caused HFTs, which were the immediate buyers, to quickly trade as they had accumulated long positions. This increased the trade volume which, in turn, increased the volume of market orders submitted by the sell algorithm. The effect of this was to cause a liquidity crisis, leading to the bottom falling from the market.

The liquidity crisis was compounded by a 'hot-potato' effect as HFTs

[2]Chronology of events taken from 'The Flash Crash: The impact of High-Frequency trading on an Electronic Market' Kirilenko et al

sought to ofload their full inventories[2]. HFTs offloaded their inventories trough trades with other HFTs. This then caused more HFTs to offload contracts, which were again brought by HFT — effectively creatign an oscillation of inventory between the HFTs in the market. Between 14:45:13 and 14:45:27, HFTs traded over 27,000 contracts – 49% of trade by volume. By the end of the period, they had only brought 200 contracts net.

The SEC blames the original sell algorithm as it took no account of timescale or price. However, this is contested by Nanex llc, a specialist firm providing information to High Frequency Traders. They analyzed the limit orders send to different exchanges in the US and found an increase in quote traffic to the New York Stock Exchange (NYSE). In the US, all exchanges report the quotes on their books to the Consolidated Quotation System (CQS) which then sends this information back to traders. The problem occured as the rate of arrival of quotes from the NYSE to the CQS exceeded the maximum processing rate for quotes from the NYSE. Thus quotes began to queue. As the quotes were only timestamped after they had been processed by the CQS, this queuing was undetectible by the traders in the market. Therefore on the CQS feeds, the NYSE therefore showed a bid price which was slightly higher than the offer price at other exchages. Sell orders were therefore routed to the NYSE, removing the buying power from other exchanges. When they were executed, the sell orders on the NYSE executed at lower prices than expected as the prices quoted in feeds from the CQS were out of date. This effect, claims Nanex, was enough to trigger the events of 6 May.

Whilst they disagree on the trigger event, the 'hot-potato' trading be-

tween HFTs is not doubted. This ate away buy-side liquidity, contributing to the rapid decline of the bid price.

2.2.1 The Hot-Potato Effect

As previously mentioned, High Frequency profits come from a very small margin on a very large number of trades. This margin is made by exploiting their ability to react before other traders. Speed is paramount. One potential threat to their profits is the changing value of the equities they hold. They therefore *want* to keep their inventory as low as possible (on the long or short side), and between strict limits. As the inventory builds, they will act increasingly desperately to manage their levels, adjusting their bid and offer price accordingly. If the inventory builds up long, they will offer increasingly competitive (with respect to other traders) offers and less competitive bids and they will start to issue sell market orders.

2.3 The Π-Calculus

The π-calculus is a process calculus which focusses on *names* — which act as channel through which processes can communicates. There are two atomic actions: a process can read a name from a channel; a process can write a name to a channel. These actions occur between two processes which run in parallel. The name which is sent through a channel may itself be used as a channel — allowing the topology of the network of processes to change.

2.3.1 Syntax

The formal syntax of the π-calcilus is as follows[18]:

π ::= x(y) [receive y along x]
| $\bar{x}\langle y\rangle$ [send y along x]
| τ [unobservable action]

P ::= $\sum_{i \in I} \pi_i.P_i$
| P_1 | P_2
| (ν a) P
| !P

The actions, π, are the building blocks of processes. The processes P, consist of several different structures. The processess $\sum_{i \in I} \pi_i.P_i$ are called summations, or sums. I in this context is an indexing set, a set of indexes i, which pick out a unique action π_i and processess P_i. We say the the processes P_i is *guarded* action π_i (alternatively, π_i *guards* P_i) as the process P_i can only execute once the action π_i has occured. We will sometimes adopt a different syntax for summations. Rather than using the sigma with an indexing set, we will used the + sign, and explicitly write the processes and actions, as follows:

$\pi_1.P_1$ + $\pi_2.P_2$ + ... $\pi_i.P_i$

Finally, the summation with an empty indexing set is the null process 0. This represents the end of a sequence of execution, but sometimes we will omit it (it is implied is an action is not followed by anything).

The processes "'P — Q"' are concurrent processes, running in parallel. The actions of these processes interleave. Processes running in parallel can comunicate with each other through exchange over channels.

$(\nu\, a)P$ represents the introduction of a new name. This new name is ony visible to the process P, so only this process can communicate over this channel. However, processes can be informed of private names, extending its scope, by senting the name of the private channel along a channel (this will become more clear after Section 2.3.2).

$!P$ is process replication. It is a shorthand for an infinite number of P processes running in parallel. It allows the π-calculus to use looping structures. In practice the processes P will be of the form $\pi P'$ where the action π guards the process. This allows us to control the replication, for example, that all of the processes but one are stalled at any one time.

2.3.2 Semantics

The semantics of the pi calculus is given by their reduction rules. This are summarized as follows:

Tau: $(\tau.\mathtt{P} + \mathtt{M}, \mathcal{D}) \rightarrow (\mathtt{P}, \mathcal{D})$

React: $((\mathtt{x(y).P} + \mathtt{M}) \mid (\bar{x}\langle\, y\rangle.\mathtt{Q} + \mathtt{N}), \mathcal{D}) \rightarrow (\{z/y\}\mathtt{P} \mid \mathtt{Q}, \mathcal{D})$

Par: $\dfrac{(P \rightarrow P', \mathcal{D})}{(P \mid Q, \mathcal{D}) \rightarrow (P' \mid Q, \mathcal{D})}$

Res: $\dfrac{(P \rightarrow P', \mathcal{D})}{(\nu\ x\ P, \mathcal{D}) \rightarrow (\nu\ x\ P', \mathcal{D})}$

Struct: $\dfrac{(P, \mathcal{D}) \rightarrow (P', \mathcal{D})}{(Q \rightarrow Q', \mathcal{D})}$ if $\mathtt{P} \equiv \mathtt{Q}$ and $P' \equiv Q'$

Tau and *React* provide the reductions for summations. As can be seen, in the event of any action reacting in a summation, the summation collapses and we are left with the process which followed the reduced action. Summations therefore present choices and the choice is resolved when the first action is reduced. They also give the reduction rules for processes which are made up of sequence of actions — as every such process can be written as a summation of the process iteself with the empty process, 0.

Par shows that the reaction of processes is not effected by other, non-interacting parallel processes. Similarly *Res* shows that if a reduction can occur in a process, this reduction is still possible isa new, resticted, name is introduced.

Finally, *Struct* says that if two processes are the same, then they react in the same way[3].

2.3.3 The Polyadic π-Calculus

So far we have only talked about single names being sent down channels. However, for many tasks we may want to send sequences of names (vectors), to allow the following reaction, for exampe:

$$\texttt{x(ab).}\bar{a}\langle\ b\rangle\texttt{.0 | }\bar{y}\langle\ cd\rangle\texttt{.0}$$

[3]Due to space considerations i haven't given a formal description of what it is for two processes to be the *same*. Informatlly it is an isomorphism of reduction — two processe are the same if there is a isomorphic function relating the reductions in one process to the reductions in another

We can give a way of encoding such actions in the original π-calculus. The *polyadic* π-calculus, which allows vectors to be sent down channels, can therefore be thought of a a shorthand for the original, *monadic*, π-calculus so that the reductions rules remain the same. Such an encoding is as follows[18]:

$x(y_1...y_n) \rightarrow x(w).w(y_1)...w(y_n)$
$\bar{x}\langle\ y_1...y_n\ \rangle \rightarrow (\nu\ \mathtt{w})x\langle\ w\rangle.w\langle\ y_1\rangle...w\langle\ y_n\rangle$

This can easily be extended to encorporate the concept of tuples. Tuples can be nested inside each other — an element of a tuple can itself be a tuple. When this occurs, the name which corresponds to the tuple will be a channel through which the tuple will be recieved (or sent). For example, receiving the tuple, $((a, b), c)$ would occur as follows:

$$x(y).y(z).z(a).z(b).y(c).0$$

Thus, in later sections, when I talk about sending tuples, it can be translated into the underlying calculus as given.

2.3.4 Abstractions

Abstractions are essentially processes with one or more unbound names. There are therefore processes which can be applied to arguments. There role in the π-calculus is to allow general processes to be defined and the used to build bigger processes. The number of unbound names in the abstraction we call the *arity* of the abstraction. Therefore, an abstraction with arity 1 is a process. Abstrations are written as follows;

$$(x\_1...x\_n)P$$

This is an abstraction of arity n. The unbound names are the names $x_1...x_n$. To apply a name to an abstraction, we use the following notation[4]:

$$P.\langle\ x\rangle$$

If the abstraction P is defined as $(y).Q$, applying x to it has the following effect[5]:

$$((\vec{x}).Q)\langle\vec{y}\rangle\ \rightarrow\ \{\vec{y}/\vec{x}\}Q$$

where the vectors $\vec{x}$ and $\vec{y}$ are of equal length. During the course of this paper in places i use an alternative sytax. The freenames in a process can be place following the process name in it's definition or proceeding the process on the right hand side. That is:

$A(\vec{x})$ `:=` Q `is equivalent to` A `:=` $(\vec{x})Q$

I prefer the first form of this, as I believe it to be more intuitive. However, in cases where the process or abstraction is being defined as acting on arbitrary processes, these are listed in parenthesis following the process name. Processes will always be given names which start with an upper-case letter,

[4]N.B. I will sometimes include commas inbetween the names for clarity

[5]The notation $\{\vec{y}/\vec{x}\}Q$ means that the vector $\vec{y}$ replaces all instances of the vector x in Q.

and channel names will always start with a lower-case letter. However, I will prefer the second form of defining an abstraction when the abstraction has arbitrary processes to prevent any confusion.

If a process or an abstraction has a process or abstraction as a parameter,for example:

```
A(B) := x(y).B
```

Then this is a different construct to an abstraction. This can be though of as a function from the abstraction or function which is the parameter to the whole abstraction or function. This is not an abstraction over processes but something prior to that. Finally, in the application of such a construct, the parameter must be an abstraction of the same arity (recall that a process is an abstraction of arity 0) as the variable parameter (in the above this is B) to be a legal construct.

This is all that is required about the π-calculus for the purposes of this paper.

Chapter 3

Analysis

3.1 Purpose and Preliminaries

In the analysis phase, I will be following several approaches to modeling financial markets. This will achieve two aims. Firstly, it will let the existing methods of modeling be evaluated against the task in hand. The strengths and weaknesses of these approaches can then be used to guide the development of a new language, which will be the body of chapter 4. It will also provide further insight into the problem at hand - showing the essential features of the markets which must be modeled.

3.1.1 Detailed description of the system to be modeled

As described in Section 2.2.1, the inventory which a HFT market-maker trader[1] holds is a source of risk to the trader. Given that HFTs are risk averse, making a small amount on a lot of trades, the accrual of inventory

[1] see Section 2.1.3 for a full discussion of what a market-maker trader is

is a key factor which the algorithm will have to prevent. There are three key factors of a HFT market-maker. Firstly, it issues both bids and offers, in an attempt to profit by other traders crossing the spread. Secondly, it will vary the size and price of the limit orders it generates, so as to return the inventory held back to the zero position (i.e. the algorithm is 'happiest' when it has no shares either long or short). Finally, the trader has limits on its inventory which, when exceeded, will cause the trader to panic, and dump its entire inventory on the market with a market order.

An algorithm with the above characteristics is then placed in the trader environment. All interaction the trader has will be done through an order-book (see section 2.1.1). The trader submits both bid and offers limit orders to the order book. These are enqueued before being processed by the order book. If the rate of orders arriving at the order-book is greater than its maximum processing capacity, there will be a build-up in this queue. This creates a delay in the system. This delays the effect of cancellations of limit orders, and can lead to greater exposure for a trader. The queue in processing means that there is a delay before any cancellations are made. This means that the actual active limit orders the trader has on the book are not what the trader thinks they are. If a traders inventory is building up, it will alter the pricing of its limit orders so as to increase the likelihood that the type of trades which will reduce its inventory will occur. For example, if the trader has a *long* inventory, it will alter its limit orders so that its *offers* are more likely to be executed than it's *bids*. However, a processing delay means that this will not occur instantly. Thus the limit order book could still have orders from the trader priced such that an increase is inventory is likely —

potentially carrying the trader's inventory over its limit. In the following sections this will be our example hypothesis for assessing the suitability of modeling formalisms, and reference to a *delay*, will be understood as referring to this delay in processing by the order-book. In practice we may want to model an order book which treats market orders and limit orders differently. Such an order book would have two different queues, one for each type of order. These queues could either be symmetric, delayed by the same amount, or asymmetric, where each queue has a different delay associated with it.

This leads us to some requirements: a suitable modeling system will be capable of:

- describing processing delays — such as a delay in processing limit orders by an order book
- describing trader algorithms (HFTs and fundamental traders)
- describing the state of a limit order book (i.e. the probability of the execution of a limit order and the price of execution of a market order)

These are the key factors the modeling formalisms will be assessed against.

3.1.2 Deciding which approaches to use

We want a way to easily describe the interaction of traders within a market. Furthermore, we want to be able to examine how different trading algorithms themselves interact within a market. The system must therefore be able to describe both the algorithms themselves and the environment in an easy way. This requires two very different methods of formal description: the

deterministic ordering of actions in the algorithm must be as easy to specify as the possible interactions of the algorithm with its environment. As described above, our example hypothesis is that delays in the system are a key factor in the hot-potato effect - thus a key part of describing the environment is to describe these delays.

I will assess two approaches to modeling this system. Firstly a mathematical, functional, approach, where the trader algorithms are described by functions and differential calculus is used to describe the markets. Secondly I will do the same with the pi-calculus. The choice of the these approaches is lead by the literature with the mathematical approach most commonly being used to describe algorithms[2] and the pi-calculus being a common formalism for describing distributed systems.

3.2 Mathematical Approach

3.2.1 A typical HFT Algorithm

In their paper, High Frequency Trading in a Limit Order Book[4], Avellaneda and Stoikov provide an algorithm for a high-frequency trader where the price of the trades issued by the HFT is dependent on a 'reservation' price - the price at which the HFT is indifferent to trade. This algorithm corresponds to the market-maker HFT strategy.

In the following section, I provide a simplified version of Avellaneda and Stoikov's algorithm. The price offered will turn out to be a function of of the current inventory held by the trader. In the version I present, this will be

[2]For example, differential calculus is used to describe the HFT algorithm in [4]

a simple, linear function, whereas the functional dependence for Avellaneda and Stoikov is exponential. This is justified as I am interested in representing the behaviour rather than giving a description of the most profitable algorithm possible.

The biggest risk a HFT market-maker faces is in the accrual of inventory[14]. Given the small profit per trade of HFTs[5] price movement in the assets held by the trader can prove very costly. To minimize this risk, the trader will always act in a way to prevent the build-up of inventory or, having failed, it will act with increasing desperation to reduce it. This will occur in two stages[4]. The algorithm will first try to reduce its inventory using limit orders and, if this fails, it will resort to market orders.

As the inventory held by the trader increases on the long side, the trader will want to sell more than it buys. Accordingly, the reservation bid price, r_b, will go down to reduce the likelihood that such orders will be lifted and trades occur. Similarly the reservation offer price, r_o will go down, to increase the competitiveness of the trader's offers to sell stock. This will make sell trades more likely than buys, thus reducing the inventory held by the trader. Given the fact that both the bid an offer prices are reduced, and given the assumption that the spread between these bids remains constant, we can represent this movement as a movement of the aggregated price of r_b and r_o called r. I assume a constant spread between these two prices, and the aggregated reservation price r is given by the mean average of the two values - that is:

$$r = \frac{r_o + r_b}{2}$$

Using this equation, and the behaviour outlined above, we can derive the

following dependence:

$$\frac{\mathrm{d}r}{\mathrm{d}t} = -f(\frac{\mathrm{d}q}{\mathrm{d}t})$$

where r is the reservation price, q is the inventory, t is time and f is a positive valued function — so that a positive rate of change of inventory leads to negative rate of change of reservation price.

The simplest solution to this is to assume a linear solution. Certainly the function will not be less than this – the HFT will not become increasingly unconcerned with the increases in inventory, and any faster moving function will exhibit the same behaviour with the introduction of delays to the system as the linear dependence produces. Thus we have

$$\frac{\mathrm{d}r}{\mathrm{d}t} = -\alpha\frac{\mathrm{d}q}{\mathrm{d}t}$$

where α is a positive constant and can be thought of as the parameter controlling how aggressively the trader acts to reduce its inventory. However, this assumes that we have a stable market midpoint — so the only factor changing the reservation price is the change in inventory. We can add the market midpoint S, into the model and we get:

$$r = S - \alpha q$$

As well as the inventory level, the size of the limit orders the trader issues will also be effected by the increase in inventory. The HFT will never issue an order which will, if hit, take it's inventory over the limit (in either the long or the short direction). This provides an upper limit on the order size allowed of:

$$s_o = q(t) - l_l$$

where s_o is the maximum size of the limit order for *offers*, l_l is the lower limit of inventory, and $i(t)$ is the inventory of the trader at time t. Similarly, we have:

$$s_b = l_u - q(t)$$

where s_b is the maximum size of the *bid* limit order and l_u is the upper limit on the inventory.

We can use this as a worst case representation of the HFT. Realistically, the size of the limit order sent is likely to be a function of many factors such as market volatility. However it will always be bounded by the above equation. Furthermore, any behaviour which occurs with limit orders given by the above equation will also occur with smaller order sizes (but will only occur when bigger delays are seen). For our purposes in creating a representation of the system, it is sufficient to use the above equation.

The two equations above provide equations for the behaviour of the HFT when it's inventory is within the soft limits, but we haven't said anything about what happens if it exceeds these limits. In this case, the trader enters a 'panic mode', offloading it's inventory via a market order. Whilst this action costs the trader money, it is still preferable to the risks involved with hanging on to the bulging inventory [14]. We will take the market order to be sized to return the trader back to zero inventory.

The HFT can then be thought of acting in two phases: normal and panic. We think of the result of the HFT algorithm as an ordered pair. The first

element is the limit order information and the second element is the market order size. That is, the result is of the form:

$$((s_o, s_b, r), (m_s, m_b))$$

where s_o is the size of the *offer* limit order, s_b the size of the *bid* order, m_s the size of the *sell* market order, m_b the size of the *buy* market order and r is the aggregated reservation price. Given the above equations for calculating the size and price of the orders, our archetypal HFT market-maker algorithm can be expressed as:

$$\begin{gathered}(q < l\_u) \wedge (q > l\_l) \rightarrow ((s\_o, s\_b, r), (0, 0))) \\ \wedge \\ ((q > l\_u) \rightarrow ((0, 0, 0), (q, 0)) \wedge ((q < l\_l) \rightarrow ((0, 0, 0), (0, q))\end{gathered}$$

This, therefore, gives us a concise way of expressing the actions of the HFT. We now go on the consider the inventory level and attempt to derive an equation for how the inventory level depends on the environment in which the trader is active. It is here that the mathematical approach starts to fail.

3.2.2 Mathematical description of the inventory level of a HFT trader in terms of market-midpoint

Background

As described in section 3.1.1, our example hypothesis for the Hot-Potato effect is that it is caused by a delay in the processing of limit orders by the

order book. This delay means that the orders a trader has on the book are actually the orders from a previous time. The time the orders come from is given by the current time in the system, minus the delay (the time the limit order spent in the queue before it was processed[3]). That is the following:

$$x = t - T(t) \tag{3.1}$$

Where t is the time parameter of the system and $T(t)$ is the latency in the system - a period of time which represents how long the processing of the queue would have to continue for, with no new members being added, in order for the queue length to be reduced to zero.

The inventory of the trader, q, is changed by making trades. Thus, the net rate of trades (buys - sells) equals the rate of change of the inventory of the trader. In this simple system, ignoring stochastic effects, the rate of trades is a function of the net imbalance of market forces (whether it is a buying or selling market), the difference between the price quoted by the trader[4] and the midpoint, and time. The price quoted by the trader will not be the current price, but due to the effect of the delay in the system, will be the price quoted at x. Thus, if $r(x)$ is the price quoted by the trader at x, S is the market midpoint and $m(S, r)$ is the number of trades which occur:

$$\frac{\mathrm{d}m}{\mathrm{d}x} = \frac{\mathrm{d}q}{\mathrm{d}t} \tag{3.2}$$

[3]See section 3.1.1 for a fuller discussion of this delay

[4]The trader will quote two prices: the sell price and the buy price. However, as we are considering only the net effect of the market on the trader,and the spread between the prices is assumed constant, the average price of the two will be all we need to show the required behavior

$$= \frac{\mathrm{d}q}{\mathrm{d}r}\frac{\mathrm{d}r}{\mathrm{d}t} \tag{3.3}$$

As previously explained, the time delay in the system occurs as there is a queue before the quotes issued by the trader are processed by the exchange. This queue length Q is therefore connected with the time delay such that, if f_r is the rate that items are removed from the queue:

$$Q = \int_{t-T}^{t} f_r \, \mathrm{d}t$$

The above equation can be rationalized as follows. Given that the latency T, is defined as the length of time which passes before an order submitted to the book is processed, integrating the rate which items are removed from the queue over this time period will give the number of items in the queue. The limits are calculated from $t - T$ to t because T itself is a function of t. When we later consider the system with a time delay, we shall only consider behavior where the system is processing quotes at its maximum rate. Thus the above equation becomes:

$$Q = f_r * T \tag{3.4}$$

As T is a function of t, this can be differentiated to give

$$\frac{\mathrm{d}Q}{\mathrm{d}t} = f_r \frac{\mathrm{d}T}{\mathrm{d}t} \tag{3.5}$$

Furthermore, we can find $\frac{\mathrm{d}T}{\mathrm{d}t}$ by rearranging equation 3.1 and differentiating to give:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = 1 - \frac{\mathrm{d}x}{\mathrm{d}t} \tag{3.6}$$

We assume that the rate of change of the queue length , Q, is a linear function of the rate of change of the reservation price, r. Intuitively this makes sense as when the price which the Trader want to trade at changes, it must issue new limit orders at this new price. Thus the faster the price changes, the faster the new limit orders must be issued and therefore the rate that quotes are added to the processing queue, Q, increases. As the rate items are removed from the processing queue is assumed a constant, this corresponds to a change in queue length. Linearity is assumed as, from the point of view of the system, this is a best case scenario: any behavior shown if this connection is linear will be made worse by other relations. Thus the rate of change of Q:

$$\alpha \frac{\mathrm{d}Q}{\mathrm{d}t} = \sqrt{\left(\frac{\mathrm{d}r}{\mathrm{d}t}\right)^2} \tag{3.7}$$

Where α is a positive constant. The equation shows that the rate of change of queue length depends only on the magnitude of the change in reservation price and not the direction. Thus the queue can grow significantly in highly volatile markets by virtue of the midpoint moving, rather than in reaction to a growing inventory. However, as we shall be looking only for a steady state solution for the system (where the rate of change of the midpoint S is a negative constant), we shall simplify equation 3.7 with the assumption that the rate of change of the reservation price r, and the rate of change of price S is always negative (i.e. in a falling market). Equation 3.7 therefore reduces to:

$$\alpha \frac{\mathrm{d}Q}{\mathrm{d}t} = -\frac{\mathrm{d}r}{\mathrm{d}t} \tag{3.8}$$

Finally, we can use the above equations to derive an expression for how the delayed time, x, depends on the other parameters of the system. From equation 3.6 we have:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 - \frac{\mathrm{d}T}{\mathrm{d}t}$$

rearranging and subbing in equation 3.5 gives

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 - \frac{1}{f_r}\frac{\mathrm{d}Q}{\mathrm{d}t}$$

then using equation 3.8 we can get

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 + \frac{1}{\alpha f_r}\frac{\mathrm{d}r}{\mathrm{d}t}$$

And finally, using equation 3.3, we get:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 + \frac{\mathrm{d}m}{\mathrm{d}x}\frac{\mathrm{d}r}{\mathrm{d}q}\frac{1}{f_r\alpha} \tag{3.9}$$

Including the rate of trades

In order to draw any conclusions from the equation 3.9 we must find an equation for the rate of limit order execution. A simple solution is to use an exponential, based on the distance from the market midpoint, S. Thus δ is the distance between the trader's limit order reservation price r and the midpoint, giving the equation for the rate of market order execution as

$$\frac{\mathrm{d}m}{\mathrm{d}t} = A\exp\left(-\delta\right) \tag{3.10}$$

Where A is proportional to the rate of arrival of market orders. This equation works for both types of trade, buys and sells, in terms of δ_b, the distance from

the midpoint to the price offered by the trader on the buy side of the book, and δ_s, the same but on the sell side of the book. The distance from the book center to the center of the two prices is given by:

$$\delta = \frac{\delta_s - \delta_b}{2}$$

and with the simplifying assumption that the distance, c between the offer and bid prices is a constant, $\delta_b + \delta_s = c$, we can get an expression for the trade rate in terms of δ, the difference between the middle of the two offered prices and the market midprice:

Using the two above equations, we can derive the following equations for the distance from the midpoint on the buy and sells sides:

$$\delta_s = \delta - \frac{c}{2}$$

$$\delta_b = \delta + \frac{c}{2}$$

The net rate of trades, $\frac{\mathrm{d}m}{\mathrm{d}t}$ is the rate of *buy* trades minus the rate of *sell* trades:

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \frac{\mathrm{d}m_b}{\mathrm{d}t} - \frac{\mathrm{d}m_s}{\mathrm{d}t}$$

Equations for $\frac{\mathrm{d}m_b}{\mathrm{d}t}$ and $\frac{\mathrm{d}m_s}{\mathrm{d}t}$ can be given equations of the same form as Equation 3.10. When the above equations are substituted in, we get:

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \exp\left(-\frac{c}{2}\right)\left[A\exp\left(-\delta\right) - B\exp\left(+\delta\right)\right] \tag{3.11}$$

where:

$$\exp\left(-\frac{c}{2}\right) = C$$

is a constant. The constants A and B are proportional to the constant rates of market orders submitted to the book on the buy and sell sides respectively.

This gives an us an expression for how the rate of trades depends on the pricing of the trader's limit order. However, due to the action of the delay in the system, this must be expressed in terms of the delayed time, x. Thus we get:

$$\frac{\mathrm{d}m}{\mathrm{d}x} = C\left(A\exp\left(-\delta\right) - B\exp\left(+\delta\right)\right)\frac{\mathrm{d}t}{\mathrm{d}x} \tag{3.12}$$

The distance, δ, is given by: $\delta = S - r$. This can be used to find $\frac{\mathrm{d}r}{\mathrm{d}q}$ — the equation showing how the trader's reservation price changes with it's held inventory. :

$$\begin{aligned}\frac{\mathrm{d}\delta}{\mathrm{d}q} &= \frac{\mathrm{d}S}{\mathrm{d}q} - \frac{\mathrm{d}r}{\mathrm{d}q} \\ &= \frac{\mathrm{d}S}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}q} - \frac{\mathrm{d}r}{\mathrm{d}q}\end{aligned}$$

Using these equations, we can derive an equation for the behaviour of the entire system:

From equations 3.9 and 3.12 we can get:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 + C(A\exp\left(-\delta\right) - B\exp\left(+\delta\right))\frac{\mathrm{d}t}{\mathrm{d}x}\frac{\mathrm{d}r}{\mathrm{d}q}\frac{1}{f_r\alpha}$$

multiplying by $\frac{\mathrm{d}x}{\mathrm{d}t}$ and rearranging will give:

$$\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 - \frac{\mathrm{d}x}{\mathrm{d}t} - C(A\exp\left(-\delta\right) - B\exp\left(+\delta\right))\frac{\mathrm{d}r}{\mathrm{d}q}\frac{1}{f_r\alpha} = 0$$

when the equation for $\frac{\mathrm{d}r}{\mathrm{d}q}$ is used, we can use this to get the equation for the behaviour of the entire system:

$$\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 - \frac{\mathrm{d}x}{\mathrm{d}t} - \frac{C}{\alpha f_r}\left(A\exp(-\delta) - B\exp(+\delta)\right)\left[\frac{\mathrm{d}S}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}q} - \frac{\mathrm{d}\delta}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}q}\right] = 0 \tag{3.13}$$

From this equation, we can derive the conditions necessary for there to be no increase of latency in the system. That is, when $\frac{\mathrm{d}x}{\mathrm{d}t} = 1$ — the values of t and x (which equals $t - T$) are constant with respect to each other, so the value of T must also be constant. For this condition be true, the following condition must also be true:

$$\frac{C}{\alpha f_r}\left(A\exp(-\delta) - B\exp\left(+\delta\right)\right)\left[\frac{\mathrm{d}S}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}q} - \frac{\mathrm{d}\delta}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}q}\right] = 0 \tag{3.14}$$

3.2.3 Problems with the mathematical approach

The equation 3.13 gives an equation for the inventory of the HFT in terms of the delay in the system. However, the underlying assumptions may drastically reduce its usefulness. In order to get the above equation, the following assumptions were made:

1. The discrete processing action can be accurately approximated by a continuous function which we can differentiate to get processing rates
2. Stochastic effects of the market can be ignored
3. The rate of change in processing queue length Q is a linear function of the rate of change of a traders reservation price
4. The rate of change of market midpoint is negative
5. The difference between the bids and offers for the trader is constant
6. The rates of incoming market orders is constant

Whilst some of these assumptions may be acceptable, the combined effect is an equation a long way away from the system it's meant to be describing.

Furthermore, the two phase behaviour of the HFTs mean that the equations for the price of limit orders is discontinuous. The above equation therefore only describes the inventory level if it stays between the soft limits. However, in order for the Hot-Potato effect to occur, the limits must be exceeded. The above equation can be used to describe a situation where the HFT inventory limit might be exceeded, showing that a stable algorithm can become unstable when delays are introduced in the system, but cannot describe the full Hot-Potato effect. One approach to extending equation 3.13 to a system of several traders interacting would be to think of the inventory levels as oscillators – two traders' inventories become coupled by executing trades. Prima facie this promises a concise solution to our problems, letting the thrashing behaviour of the 'hot-potato' effect be described in a simple, intuitive way. However, due to the non-differentiable nature of the two-phase HFT algorithm given by the equation at the end of Section 3.2.1, this interaction is highly complex and defies simple analysis. We therefore require another method of describing the interaction of different traders, for which we use the pi-calculus.

3.3 Modeling using the π-Calculus

The pi calculus's strength is in modeling how processes running concurrently can interact with each other. This is exactly what we want to model in the market. Each trader is a process holding state which operates in a loop, responding to changes in its environment by changing the environment. At a simple level, where we shall start, the environment is just a single order-

book with the associated trade matching engine and quote reporting system. For this high level, the pi calculus therefore seems ideal. The high-level description of the system is very easily achieved in the pi calculus. For example, three traders interacting through an order book can be given as follows:

```
(ν in) ( trader1(in) | trader2(in) | trader3(in) | orderBook(in))
```

Here the three traders and an orderBook process are running concurrently. The bound name (in) is the channel through which the traders can send orders to the order book. The traders and order book are all considered to be abstractions with arity one (see Section 2.3.4) and are thus each applied to the same name, linking them.

However, this says nothing about the action of the order book or the traders themselves – the algorithms by which they run. The suitability of the pi-calculus for our purposes then, will rest on its ability to describe the algorithms themselves. The pi-calculus is Turing complete[18], so it is possible to encode any algorithm with it. However, we want the description to have the following characteristics

1. It should be easy to encode
2. Once encoded, it should be easy to understand
3. Once encoded, it should be easy to change

The rational for the first condition is obvious. As the system is intended as an aid to research, it should make the researchers job as easy as possible

(otherwise it's a poor aid!). The description should be easy to understand to facilitate communication amongst researchers ideally someone who has not seen the pi-calculus (or been given a very quick introduction) should be able to understand it (or at least get the gist). Finally, as research is a process, the descriptions should not have to be re-written substantially in order to amend the algorithms slightly.

Keeping these three conditions in mind, I now move on to implementing the algorithms in the pi calculus. An essential part of these are the data-structures. The order book is essentially a list of queues of limit orders. Thus it seems that both list and queues should be easy to implement for the pi calculus to be suitable.

3.3.1 Data-structures in the π-calculus

In his book [18], Milner shows how complex data-structures can be built up in the pi calculus using the idea of switches and nodes. A switch is an abstraction which is given a list of options and choose one of them[5]. Nodes are single storage elements, storing one item (although this itself could be the name of another data structure). Lists can be formed by making each node hold two things: the data to be stored and the name of the next node. The switch fits in, as each node is sent two names *nil* first and *cons* second. The node returns a message along the name corresponding to nil if it is the last element in the list. Otherwise it returns an ordered pair containing the name of the data-item, and the name next node in the list. The last node is therefore a special case we'll call *Nil* and all the other nodes we'll call *Cons*.

[5]This is the same idea behind conditionals — see Section 3.3.2

Milner defines these as follows[18]:

$$Nil \texttt{ := } (k).k(nc).\bar{n}\langle\ \rangle$$
$$Cons(V, L) \texttt{ := } (k).(\nu\ \texttt{v l})(Node\langle\ kvl\rangle \mid V\langle\ v\rangle \mid L\ \langle\ l\rangle)$$
$$\texttt{where } Node(kvl) \texttt{ := } k(nc).\bar{c}\langle\ vl\rangle$$
$$\texttt{[}V\_1, ..., V\_n\texttt{]} \texttt{ := } Cons(V\_1, (Cons...(Cons(V\_n, Nil)...))$$

Here, the process V which is being stored is accessed via the name v. The rest of the list, L, is stored in parallel with the the $Node$ and data processes.

This implements a list in the functional programming sence. Other data-structures are similarly possible, and in Appendix D I give my own implementation of a queue. However these are difficult to follow. Even the last part of the above definition (all taken from [18]) is confusing and it itself is a shorthand. When it is compared to a definition of the same structure in Haskell, this problem is more clear[6]:

```
data List a = Cons a (List a)| Nil
```

The extra names and odd composition (the data is in i processes in parallel, even though we want to think of it as a seqeunce) of data-structures in the π-calculus means that a simple structure is difficult: both to construct and understand. We would therefore like an easier way of dealing with data-structures in the language we use for the modeling of markets.

[6]It may seem unfair to be comparing a high-level language, Haskell, with a low-level calculus. Indeed, the implementation of a list in the λ-calculus is likely to be similarly ugly. However, I am not giving a complete assessment of the π-calculus, but am evaluating how it can be applied to a specific task. Therefore, as we would need data-structures to model what we want to model, we need this to be easy to do in the π-calculus.

3.3.2 Defining Conditionals in the π-calculus

In order to encode any kind of strategy, we need a way of choosing between different possible actions – we need conditionals. In the pi calculus, choice is achieved with the summation construct, which is defined as follows[18]:

$$\texttt{P ::=} \sum_{i \in I} \pi_i.P_i$$

Where I is an indexing set. The action π_i is guard on the process P_i. This means that, in order for process P to run, the action guarding it must execute. There is an alternative syntax for summations which, in some cases, is easier to understand:

$$\texttt{P ::=} \pi_1.P_1 \ + \ \pi_2.P_2$$

This is equivalent to the first notation where the indexing set contains the same π_1 and P_1 and π_2 and P_2 as in the second example. In practice, the second notation is the easier to follow, and I shall therefore use this in preference to the first.

When a summation executes, one of the actions preceeding the processes (i.e. one of the guards) executes. The summation then collapses to the process which follows the reduced action. For example, the following reaction:

$$\bar{x}\langle \ y\rangle.0 \ \mid \ x(z).P_1 \ + \ y(z).P_2 \ \rightarrow \ \{y/z\}P_1$$

The notation $\{y/z\}P_1$ means that the name y replaces all occurances of the name z bound by this binding construct[7] in the process P_1.

This is the essence of choice. Thus, if we want to create a basic conditional where one of two actions happens depending on the name sent to the process, we have:

$$in(name).n\bar{a}me\langle\rangle.0 \texttt{ | x().P1 + y().P2}$$

where $P1$ is the process executed if x is sent, and $P2$ is the process executed if y is sent. This is easily extended to include an encoding of boolean values[8]. Booleans can be encoded as abstractions[9], which are sent two names and pick one of them. *True* picks the first of these options, and *False* picks the second:

$$\texttt{True(l) := } l(tf).\bar{t}\langle\rangle.0$$
$$\texttt{False(l) := } l(tf).\bar{f}\langle\rangle.0$$

This idea of boolean can be used with an amended form of the conditional given above, which shows how boolean values can be used with conditionals:

$$\texttt{Cond(P,Q) := (l)}(\nu \texttt{ t f})\bar{l}\langle\ tf\rangle.\texttt{(t.P + f.Q)}$$

[7]If there is an inner process which rebinds the name z then these are left as they are — by convention if a name is in the scope of more than one binding construct, we use the most specific.

[8]All of the processes and abstractions in the definition of booleans are taken from [18]

[9]See section 2.3.4 for a discussion about abstractions

where P and Q are processes.

Cond is an abstraction with one free name — l. It acts to choose between the processes P and Q. When this process acts concurrently with the booleans, we get the following reactions[10]:

$$True \ \langle \ \ l\rangle \ \texttt{|} \ \texttt{Cond}\langle \ l\rangle \ \rightarrow \ \texttt{P}$$
$$False \ \langle \ \ l\rangle \ \texttt{|} \ \texttt{Cond}\langle \ l\rangle \ \rightarrow \ \texttt{Q}$$

Simple choices, then, are simple. However, the above abstractions do not address the issue of equality testing, and it is here that the difficulty emerges. If there is not an action which can execute in a summation, then the process hangs. Thus, when creating the conditional test, care must be taken to make sure that there are processes for all possible actions, otherwise there is a probability that the process will hang. Thus inequality testing requires processes for all values that the test name can be[11]. This problem of making sure the summation covers the entire domain is significant – making equality testing over a domain of names very difficult, as for each name in

[10]Note that the Conditional absrations below must have been given specific processes P and Q which are constituent parts of it. I use the general terms below, but these are concrete processes

[11]As an aside, this may seem to make the comparison of numbers impossible as they require an infinite summation. However, as will be seen in the next sub-section, a number is encoded as a sequence of names thus the comparison of numbers is a sequence of comparisons with domains of two names. This sequence will be finite as we are not able to encode infinite numbers. However, when we extend the π-calculus to include numbers and arithmetical operators, we will have to add equality operators to solve this problem

the domain, there has to be a corresponding process in the summation[12]. This is a major weakness, making the pure pi calculus difficult to work with, breaking condition 1 in Section 3.3. This must be simplified in order to adopt some variant of the pi calculus as our modeling language.

3.3.3 Doing Arithmetic in the π-Calculus

The pure pi calculus contains no numerals or mathematical constructs. In order to do arithmetic, we therefore need a way to encode these in the pi calculus, or else it must be extended with integers and operators (which may require typing). Integers can be encoded using a method very similar to church numerals in the lambda calculus[18]. We can think of a number as sequence of names. These names are either e, zero in the church numeral system, or m which is the successor function in church numerals. The cardinality of the number encoded is then given by the length of the sequence of m names. Furthermore, it is meaningless to have a sequence of ms which do not terminate in an e. This gives us the following:

```
0 = e
1 = me
2 = mme
...
```

for which we can use the shorthand

[12] As example of this problem is removing all members of a queue with a given name. See appendix E for an example of how this can be done

$$\texttt{n} = m^n e$$

Having created a way of encoding integers, we need to define the operators for arithmetic. I will only define the addition operator but the other operators can be defined by a similar method[13].

Addition is easy. I define a process with two inputs channels and one output (although it could just as easily be done with only two or even one channel). Along each input channel, a number encoded by the above method will be sent, and the result will be sent along the output channel. To add the two integers, the process just has to read from one channel, and send it back out along the output channel until it reaches the e. This it does not send back out, but switches to reading from the second channel sending the complete second integer out on the output channel. That is the following:

$$\begin{aligned}&\texttt{Add(num1,num2,out,m,e) := } (\nu\ \texttt{sem sem2})\ (s\bar{e}m\langle\ \rangle \\ &\texttt{ | } !(sem().num1(x).\bar{x}\langle\rangle.0\ \texttt{ | }\ m().o\bar{u}t\langle\ m\rangle.s\bar{e}m\langle\rangle.0\ \texttt{ + }\ e().s\bar{e}m2\langle\rangle.0) \\ &\texttt{ | } !(sem2().num2(x).\bar{x}\langle\rangle.0\ \texttt{ | }\ m()o\bar{u}t\langle\ m\rangle.s\bar{e}m2\langle\rangle.0\ \texttt{ + }\ e().o\bar{u}t\langle\ e\rangle.0))\end{aligned}$$

However, this still has problems. In the naive representation given above, the names m and e must be bound globally – thus are visible to the entire program. This creates a problem when when doing the comparison to check whether a member of the input integer is either a m or and e. As these names have global scope, if there is more than one addition process operating

[13]I define addition as it is the simplest — multiplication is defined recursively in terms of addition, see [1]

in parallel, they can interfere with each other. There are possible solutions to this. Integers could be amended to have unique names. In order for the processes operating on these new style integers, these two name must be communicated first. Thus an integer becomes the following abstraction:

$$
\begin{aligned}
I_0(m,e)(l) &:= \bar{l}\langle\ e\rangle \\
I_1(m,e)(l) &:= \bar{l}\langle\ m\ \ e\rangle \\
I_2(m,e)(l) &:= \bar{l}\langle\ m\ \ m\ \ e\rangle \\
&\cdots \\
I_n(m,e)(l) &:= \bar{l}\langle\ m^n\ \ e\rangle
\end{aligned}
$$

These can be placed in a context where they are accessed along a channel which supplied the two names m and e as well as the name of a return channel. For readability, we will assign the name of the channel to the numeral which the integer process represents. This should not be confused with the actual numeral, it is just a standard name. All of these standard names are bound globally. Doing this for arbitrary n, we have[14]:

$$N\_n := !(n(m,e,l).I_n\langle\ m,e\rangle\ \langle\ l\rangle$$

where n is a place holder for an arbitrary numeral and N_n is the name of a number process. For example, encoding the number two:

$$N\_2 := !(2(m,e,l).I_2\langle\ m,e\rangle\ \langle\ l\rangle$$

Using this new definition of a numeral, we have to amend the definition of addition, to get:

[14]The notation below is the application of names to an abstraction. See section 2.3.4 or [18] for more information

$$\texttt{Addition(num1,num2,out) :=}$$

$$(\nu\ \texttt{m e n1 n2})\ \bar{n}1\langle\ m\ e\ n1\rangle.n\bar{u}m2.\langle\ m\ e\ n2\ \rangle.Add\langle\ n1\ n2\ out\ m\ e\rangle$$

This returns the sum of the two numbers as a church numeral. The role of the integer abstractions is therefore at the beginning of a calculation — they make sure that the names for m and e are unique for that calculation. The above abstraction is by no means ideal. If operations are to be sequences, there will have to be a way of passing on the unique names. A simple way of doing this would be to build a process similar to the integer abstractions themselves, but named as the result of the calculation. Such a process can be done using a queue abstraction similar to that defined in section 3.3.1 and is given in appendix C. Together these processes allow addition to be done in the pure π-calculus, but they add a further complexity to an already convoluted system.

Thus the pi calculus is unsuited to arithmetic. Although it can be done, even basic operations are tricky to define, making even integer arithmetic breaks the first of the requirements outline above (section 3.3). Furthermore, integer arithmetic is probably not enough we will likely require some kind of floating point representations. This could possible be avoided given the fact that both shares and ticks are discrete values thus trade values for a single order-book are discrete. We could therefore encode the pricing such that the tick size was a single integer meaning that all trades would be of an integer quantity. However, the trader strategies are likely use continuous functions which are then rounded in calculating the pricing of their limit orders. These functions will then be rounded to the nearest tick to produce

the final value of the limit orders. For calculating these continuous functions, we want higher precision numbers. Therefore it is best to include floating point numbers to allow for this eventuality.

This does not mean we should turn our back on the pi-calculus. Given the ease with which the concurrency aspect can be encoded, this would be foolish. However, we should reject the pure pi calculus, instead choosing to extend it with rational numbers and arithmetic operators.

3.3.4 Controlling the proliferation of names

Another factor against the pi calculus is that the proliferation of names makes all but the most simple programs difficult to follow. For a program to be syntactically correct, all of the names within it must be bound[15] - an unbound name is meaningless. However, the pi calculus has a variety of binding constructs, which can add to confusion. Names are bound both by restriction, and by reading action processes (example: $x(y)$ binds the name y). This is fine for simple programs, but for larger, more complex constructs it starts to become confusing.

It becomes confusing for two reasons. Firstly, programs cannot be organized so that the binding of names at each level is done together. Action processes must be ordered in the correct way (the order the actions must take place in!) so finding the binding for a given name can be difficult.

In addition, each reducible pair of action processes (i.e. each *in* and *out*

[15]We can have an unbound name in an abstraction (see Section 2.3.4), but this is only to allow the definition of a parametrized process. Abstractions in use must be supplied with the appropriate bound names

pair) binds a new name. Thus, for every reaction, there is a new name introduced. This creates a problem keeping track of all of the names introduced so as to avoid overriding any binding, and thus changing the program semantics – a problem similar to β-reducing in the lambda calculus and accidentally binding a previously unbound name. This problem can be averted by carefully defining abstractions. This way, programs can be broken into smaller building blocks. The unbound names in an abstraction allow it to be a constituent part of a larger process, and will typically be the input channels to the abstractions. Abstractions can therefore be thought of as functional black boxes. This reduces the impact of the proliferation of names by limiting the scope of names to the sections where they are relevant. This, however, causes a further potential problem.

Using abstractions like black boxes means that for each abstraction, we need a protocol of how to send data to the abstraction. A simple example would be an abstraction which carries out a simple operation – subtraction. If this abstraction is given only one input channel, then we need to know whether the minuend or the subtrahend is sent first. A similar problem occurs if we give it two input channels – which channel is which? As these abstractions get more complicated, the protocols required become more complicated. To reduce this problem, we can standardize these abstractions and use a notation where the protocol is implicitly communicated. Thus this problem will have to be addressed if we intend to use the pi-calculus as our notational system.

3.3.5 Replication and Recursion

As mentioned in section 2.3, the pi-calculus does not allow recursive definitions, as replication can be used instead. For many tasks this doesn't matter, and indeed for some applications this may be an easier way to reason about parallel processes, however when it comes to encoding algorithms, recursion is often easier to follow for two reasons. Firstly, the replicative definition requires the use of names and reductions as semaphores — increasing the complexity of the equivalent definition. Secondly, replication and recursion chop-up program flow in different ways. For some algorithms, the natural program flow could lead to shorter and more expressive definitions when recursion is used rather than replication. Therefore, to aid the readability of the language, and to add to the ease with which trader strategies can be encoded, a system which allows recursive definition is preferable over one including only replication.

3.3.6 Keeping track of time

Given our hypothesis of the importance of delays for the occurrence of the Hot-Potato effect, it is essential that we be able to simulate the passage of time in the π-calculus. We can do this using a timestamping technique. The algorithms work in loops: the order-book runs a loop which takes an order from an input queue and processes it. The traders run loops where their state is assessed, and then limit orders issued. Time stamping can work by include a third timing process. Each timed process (order-book or trader) is given a unique guard on each loop. Also at the end of the loop, it sends an empty

message along this channel. At the beginning of each time period, the timing process sends out message along all of these guard channels, activating all of the processes. These processes then execute and send the guard name back to the timing process. At some point, provided the process does not hang, the timing process receives all of the names back. This marks the end of the timing processes timestamp, and the whole process repeats itself[16]. By this method, time can be kept track of, if at the end of each timestamp, the timing process sends a message to a counter process.

This shows that it is possible to work in a notion of time to the pure pi calculus. However there are many reasons why we might not want to do it. It works, but is clunky and inflexible. It is also hard to extend to allowing different processes to be timestamped by a different amount: a situation where a trader loop is a fraction of the time taken for an order-book processing loop. It is also very difficult to define a signal delay[17]. This inelegance also distracts from the actual process which we want to examine. This is another aspect which should be corrected in the final language.

[16]I.e. there is an action which stalls a process until every process in the program has completed all the actions for a particular timestamp. Only then do these guard actions become reducible, and the program can continue

[17]The signal delay would have to be modeled as a complete timestamp and then this timestamp unified with other timestamps in the system (i.e. the processing loop timestamp include ten delay timestamps)

3.4 Stochastic π-Calculus

Although it has promising aspects, the pure pi calculus is not suitable for the task at hand. However, there are many variants of the pi calculus one of these may be suitable. One of such is the *Stochastic π-calculus*. This combines the pi calculus with a stochastic reduction order. An advantage of this is the existence of simulators, such as SPIM[18] which take code, compile it down to the Stochastic π-calculus, execute it and provide visual feedback of the result (plotting a graph etc). In addition, the high-level language provides a type-system, operators and data structures preventing many of the problems encountered in our exploration of the pure π-calculus. This therefore demands further examination. One key question to be answered is whether the reduction semantics is suitable for what we want.

3.4.1 Reduction semantics of the Stochastic π-Calculus

The Stochastic Pi Machine, or SPIM, is intended to be used in the modeling of chemical reactions. The variant of the Stochastic Pi-calculus which it runs is therefore optimized for this task. The semantics is based on the idea of rate environments. Every process P in the stochastic pi-calculus has a class μ of *measures*; for each action α in P there is a defined rate for that action. The environment is therefore the set of rates defined over the channels in P [7]. These rates represent the reaction rates of channels the rate of reaction between reactants in a chemical reaction. In the abstract machine for the pi the Stochastic Pi Calculus, the next channel to reduce

[18] Available at http://research.microsoft.com/en-us/projects/spim/

in the program is calculated using Gillespie's algorithm (Exact Stochastic Simulation of Coupled Chemical Reactions), creating a system capable of easily simulating chemical reactions.

However, this introduces a level of indeterminism that we don't want. We do not want the algorithms used to have a stochastic rate where the time of a reaction is stochastically determined so that the mean time is equal to the rate[13]. We want the rate of processing to be a constant. This is much closer to how the algorithms actually operate — they are not inherently stochastic. Therefore SPIM is not suitable for our task.

SPIM can still be of use, however. The type-system and included operators could serves as a model when creating our own system.

3.5 Final Thoughts on Analysis

During the analysis phase, we have examined three approaches to modeling a market with HFT traders and have discovered the limitations of these approaches. These can be summarized as follows:

Limitations of the mathematical approach:

- The algorithms involved are discontinuous. Equations describing the rate of trades (for example) must therefore be bounded in regions where the algorithms behaviour is continuous. Therefore, rather than one equation to describe a system, we need to have a different equation describing the system in each of the continuous regions.

- The order book is not modeled directly, but modeled as an equation giving the liklihood that limit orders are executed given their price.

This model requires a large list of assumptions which drastically reduce its usefulness.

Limitations of the π-calculus approach:

- Arithmetic is difficult and prone to bugs
- Complex prcesses lead to a proliferation in the number of names which makes than difficult to follow, and error-prone
- It uses replication over recursion which is nt suited to the expression of trader strategies
- Although the passage of time can be marked, it is inflexible and therefore unsuited to describing systems in which time delays are important

The stochastic pi calculus, although it seemed promising, was rejected as the reduction semantics were not suitable for the modelling of the markets.

In the next section, I use this gained knowledge to create a new language which allows a functional approach, to allow the easy specification of algorithms, within an altered pi-calculus, which allows the specification of delays in the system. Due to the limited scope of this project, I do not provide formal proofs of bisimularity[19] within this new system, nor a full implementation, but give an informal description of what it means within this new delay-calculus.

[19]Bisimularity between process is (loosely) the property of exhibiting the same behaviour - see [18]

Chapter 4

Design

4.1 Preliminaries

In the following sections, I give a description of a new language (which I have named Spiranda) for modeling markets. It is based on the findings in the analysis section and unifies a mathematical-functional approach with an extended pi-calculus. First, I give the requirements for this system.

4.1.1 Requirements

From my work in the analysis phase, I found that there were five important characteristics which the system should meet.

1. Numerical Alogrithms should be easily expressed. This means there should be some facility for recurssive definitions.

2. Agents operating concurrently should be easily expressed

3. Delays should be easily described and it should be easy to change the delays in an existing system

4. There must be a simple way of storing information

5. The system should be easy to learn to use

4.1.2 Methodology

In order to create a system which meets the above requirements I define two layers. I extend the π-calculus to include delays, arithmetic and basic data-structures (lists and tuples). This meets the requirement 2, as the π-calculus excells at this kind of description, requirement 3 as this new delay syntax is both easy to use and alter and the addition of arithmetic and data-structures helps requirements 1 and 4. Above this, I define a higher language which includes elements of the functional programming paradigm together with a translation down to the delay pi-calculus. This higher-level language therefore acts as syntactic sugar, but in doing so provides a mechanism for expressing functions with recurssion, meeting requirement 1. The success or failure of the system in meeting requirement 5 will depend on how easily the conversions can be done.

4.2 The delay pi-calculus

The delay pi-calculus is the pi-calculus extended with a new atomic action – the delay. The motivation for this is the need for a meaningful way to measure the passage of time in our modeling language. Early research[17] showed

the effect which processing delays can cause on a market. Our example hypothesis also has processing delays as the cause of the *Hot-Potato* effect[1]. Thus our language should be able to express this.

4.2.1 Syntax

The syntax is given as follows.

```
P ::= π.P
    | P|P
    | !P
    | (ν x)P
    | Σ
    | if x then P else Q
    | 0

Σ ::= π.P + Σ
        | π.P

π ::= x(y)
      | x̄⟨ y⟩
      | δ

δ ::= d_i[l]
```

[1]See Section 3.1.1

All grammar items have the same meaning as that outlined in Section 2.3 but it contains two new construct, the if operator and the new atomic action δ.

The use of the if operator is adapted from [6] and it is included to simplify conditionals in the calculus. Its reduction semantics are given in Section 4.2.3.

The delay, δ is of the form $d_i[l]$ where l is an integer defining the length of the delay and d_i is a unique identifier for the delay. There are no bounds on the size of l, nor implicit units for the length. Rather, the integer defines a passage of time relative to other delay times defined in the same program. Thus it can be used to encode behavior and systems across any time-scale - as long as all the times are encoded correctly relative to each other. For example, given a need to encode two delays, one of of 100 milliseconds and another of 10000 milliseconds, we can use the following actions:

$$Delay_1 = \delta[1] Delay_2 = \delta[100]$$

Provided there are no other delays in the system, this is identical to the following[2]:

$$Delay_1 = \delta[10] Delay_2 = \delta[1000]$$

i.e. it is not the absolute values encoded which matters, but the ratios between the values.

[2]these are only identical if there are no other delays in the system — see Section 4.2.3

4.2.2 Delay Environments

A delay environment E is a tuple consisting of the set of encountered delays in a process, $\mathcal{D}$ together with the set of all possible standard reductions, $\mathcal{R}$.[3] in a process. A reducible delay is a delay in unguarded position in a process[4]. The set of encountered delays, $\mathcal{D}$ for compound statements can be build up using the following recursive function, *delays*.

```
delays (P|Q) = (delays P) ∪ (delays Q)
delays (π1.P + Q) =
      x | x ∈ D ∧ ∀ y (y ∈ D → (Shorter x y ∨ Equal x y))
     where D = (delays π_1) ∪ (delays Q)
delays (!P)  = delays P
delays ((ν x)P) = delays P
delays (x(y).P) = ∅
delays (x̄⟨ y⟩.P) = ∅
delays (δ.P) = {δ}
delays 0 = ∅
```

The predicate *Shorter* is true if the length of the first argument is less than the length of the second argument, and the predicate *Equal* returns true if the two arguments are the same length. The function for summations

[3]Standard reductions are the reductions of the pure pi-calculus. See section 2.3

[4]Unguarded position corresponds to the first action in a process. If the process is a parallel composition of many processes, there will be many 'first' processes — hence the different terminology. The key point is that only actions in unguarded position are candidate for reduction

returns the shortest delay from all of the terms in the summation. This is because, in this context, only the shortest delay will possible execute[5].

This set, $\mathcal{D}$ as defined by the function above, is used in the definition of the reduction rules in the delay π-calculus.

4.2.3 Reduction Semantics

The reduction semantics for the delay pi-calculus is as follows. The reduction rules for actions other than delay actions is the same as for the pure pi calculus. The reduction of delay actions is as follows. Any delay of length zero can be reduced. If there exists any delay of length zero, then no standard reductions can be performed (i.e. a delay of length 0 must be reduced before any standard reductions occur — this assures that equal length delays are reduced together). If there are any standard reductions which can be performed, then no delay reduction can be performed. If there are no standard reductions, then the delay with the shortest length can be reduced. This reduction produces the process with the delay action removed, and a new environment. This new environment is calculated from the old environment by reducing the length parameters on all of the remaining delays by the length of the delay which has just been reduced. The program to be reduced is a tuple containing the process and the delay environment.

The standard reductions expanded with the if reduction[6] and altered to include the delay environment:

[5]This will become apparent when I give the reduction semantics — Section 4.2.3

[6]corresponds to the *Match* rule given in [6]

$$\texttt{Tau: } (\tau.P \texttt{ + M},\mathcal{D}) \rightarrow (\texttt{P},\mathcal{D})$$

$$\texttt{React: } (\texttt{(x(y).P + M) | } (\bar{x}\langle z\rangle\texttt{.Q + N)},\mathcal{D}) \rightarrow (\{z/y\}\texttt{P | Q},\mathcal{D})$$

$$\texttt{Par: } \frac{(P \rightarrow P',\mathcal{D})}{(P \mid Q,\mathcal{D}) \rightarrow (P' \mid Q,\mathcal{D})}$$

$$\texttt{Res: } \frac{(P \rightarrow P',\mathcal{D})}{(\nu\ x\ P,\mathcal{D}) \rightarrow (\nu\ x\ P',\mathcal{D})}$$

$$\texttt{Struct: } \frac{(P,\mathcal{D}) \rightarrow (P',\mathcal{D})}{(Q \rightarrow Q',\mathcal{D})} \texttt{ if P} \equiv \texttt{Q and } P' \equiv Q'$$

$$\texttt{If: } \frac{(if\ x\ then\ P\ else\ Q,\mathcal{D}), x = true}{(P,\mathcal{D})}$$

$$\texttt{Notif: } \frac{(if\ x\ then\ P\ else\ Q,\mathcal{D}), x = false}{(Q,\mathcal{D})}$$

Delay reductions:

$$\texttt{Delay: } \frac{(d[l].P,\mathcal{D})}{(P,\{d_i[l_i] \mid\ d_i[(l_i+l] \in (\mathcal{D} - \{d_i[l_i]\})\})}$$

The rationality for these semantics is the separation between actions, and time. Standard reductions can be thought of as occurring in zero time, thus any standard reductions will take place before any delay reductions do. Although this might seem to depart from reality but this separation provides a greater flexibility to the system. Non-zero time standard actions can be encoded by placing a delay before the action which takes the time. Alternatively, if there is a process which takes an input, does some processing and then sends a name out on an output channel, then this can be modeled by placing the delay anywhere between the input and output actions.

4.2.4 Adding Arithmetic

We want it to be easy to do arithmetic in the new system. To enable this, the delay pi-calculus is extended with rational numbers and arithmetic operators. We also want to simplify conditional testing in certain cases. I therefore add booleans and the triadic *if* operator. This extenstion demands the defininion

of two things: the context where these new construct can be written, and the reduction rules for the constructs.

To help define the context, we define an *value*. A value is an atomic constant *value*. This is defined as follows:

```
value ::= name | num | bool
```

Along with the values, we added operators to the language. These constructions form *expressions* and expressions can be passed along channels. We therefore have to define the syntax for expressions, and amend the definition of the actions to allow the passage of expressions, rather than just names[7]:

$$\pi \texttt{ ::= } x(exp) \texttt{ | } \bar{x}\langle\ exp\rangle \texttt{ | } \delta$$

```
exp ::= value | exp Bop exp | unop exp

Bop ::= '+'
      | '-'
      | '*'
      | '/'
      | '^'
      | '='
      | '<>'
      | 'And'
      | 'Or'
```

[7]The following describes the the legal constructs of the language, but does not require that these construct respect a typing

```
Unop ::= 'not'
```

Finally, we need to define the reduction rules for values. These are the same as δ rules added to an extended untyped λ-calculus[8]. These are the usual rules of arithmetic recursively defined[9].

4.2.5 Working with Lists

In order to be able to think, and program, in the functional style we need to be able to work with recursive lists. There are two options for this, we can give a way of encoding lists in the *pi*-calculus or we can add lists as a value. Both options are easy to achieve Robin Milner gives a way to encode lists in the pi calculus in his introduction to the *pi*-calculus [18]. Nevertheless, I think it is more intuitive for a list to be a value, and thus added as a new object to the calculus. The reason for this is because it closer matches how they will be used. They are a container for other value types and as such, it seems reasonable that they should be the same *kind* of object as the objects which the hold.

Thus we add the list structure to the definition of a value given in Section 4.2.4. As the list type is a compound type, we amend the definition of a value slightly. We also have to add the two list-operators *head* and *tail*. Thus we have the following[10]:

[8] lecture 5, UCL Functional Programming course

[9] I do not do it here as it is extensively covered elsewhere — see any introductory text to Peano arithmetic

[10] The operators Head and Tail work on values here, as they can be applied to Names which stand for values of list type. The definitions here are the legal constructs and as such do not respect a typing

```
value ::= name | num | bool | list
exp ::= value | exp bop exp | unop exp
bop ::= '+'
      | '-'
      | '*'
      | '/'
      | '^'
      | '='
      | '<>'
      | 'And'
      | 'Or'
bnop ::= 'not' | 'head' | 'tail' | 'length'
list ::= value ':' list | '[]'
```

This gives us recursively defined lists in the system which can easily be passed around as values.

4.2.6 Type-system and Inference

The Problem

The introduction of *values* into the language has introduced the possibility of run-time errors. These are not to be confused with problems such as deadlock, livelock or starvation which may occur in the pure π-calculus these are not errors but valid outcomes, essential when investigating the systems. The runtime errors are due to attempting to use values in the the wrong way: they are type errors. For example, names, integers and expressions can all

be sent along channels. However, only names can act as channels. Imagine a process which receives a value along some input channel, uses this value as a channel and sends out another value along it. There will be a runtime error if an integer is sent along the input channel. That is, the following process will give a runtime error:

$$(\nu\ \mathtt{x})(\ x(y).\bar{y}\langle\ 2\rangle.0\ \mid\ \bar{x}\langle\ 1\rangle\ )$$

While I said the introduction of values has caused this problem it isn't necessarily so. We could choose to allow the above process, allowing all values to act as channels. This is somewhat akin to using a weak-typing system. While prima facie this might seem to solve a problem, I believe is completely unnecessary and, what is worse, will make the system harder to use. Names and integers are fundamentally different so much so that it is somewhat of a wrench to include integer in the π-calculus at all. As such, there should be no reason to ever want to make an integer or expression act as a name. If it occurs, then, it is probably the result of an error. Not allowing this to occur will mean that any errors are explicitly shown up. Furthermore, with a suitable type-system, these errors can be caught statically (i.e. at compile time) reducing the change of runtime errors in the system: a program containing such an error will not compile. This is the motivation for introducing a strong static type-system.

Milner's Sorts and Sorting

The type-system I introduce is based on that outline in Milner's introductory book on the π-calculus[11]. I do not give *delays* types because the interact with nothing (other than a runtime system) types tell us what can interact with what.

Milner's type-system is based on Curry-style typing[12]. An important characteristic of this approach is to allow polymorphic functions. Where Church-style typing (the other classical approach) would treat a polymorphic function as many distinct cases, one for each case, Curry-style typing allows the intuitively simple notion of one term being able to be applied to many different types.

For Milner, a typing of a process is a partial function, called *ob* from the collection of types, Σ, to the set of all type-lists $\Sigma^\dagger$. A type list, σ is just a finite sequence of types. The function *ob* is understood as defining the types which can be carried by the type it is applied to. We say that a processes is well-typed if all sub-terms of the form $x(\vec{y}).P$ or $\bar{x}\langle\vec{y}\rangle$ in the process respect the condition:

$$\texttt{if x:}\sigma\ \texttt{then}\ \vec{y}\ \texttt{: ob(}\sigma\texttt{)}$$

The ':' here means *is of type*. The above expressions essentially says that a process is well typed if every time something is sent down a channel, it is of an allowed type for that particular channel.

[11] He calls this a system of *sorts* but it amounts to the same thing: [18]

[12] see [1] for a discussion of different ways of typing

We say a system is well-typed if such a typing can be given. The above definition was given by Milner as a general approach to typing in the pure π-calculus. He goes on to extend this system by adding compound types. A type is now defined as a constructor C followed by a perhaps-empty list of types $\vec{\sigma}$. Each compound type constructor (i.e. a type which include other types as constituents) is given a typing rule, in the following format:

$$\texttt{ob : C(}s_1\texttt{,...,}s_n\texttt{)} \mapsto \rho\_1 \texttt{ ,..., } \rho\_n$$

Here, $s_1, ., s_n$ are type-variables, which may or may not appear in the types $\rho_1, ..., \rho_n$. Thus, the function *ob* is calculated for the type $C(\vec{\sigma})$ by substituting in the types $\sigma_1\sigma_n$ for the types $s_1, , s_n$.

The most important of these constructors is the *Channel* constructor. This is given the following rule for the function *ob*:

$$\texttt{Channel (}\vec{s}\texttt{)} \mapsto \vec{s}$$

This forms the backbone of the type-system we shall adopt.

Adapting Milner's Sortings

Firstly, I slightly alter Milner's approach to bring it closer to Church typing in the lambda calculus[1]. I do this as most of the construct I will be providing typing rules for are operators. I therefore define each construct which can have a type, in terms of the type it takes. This gives us the following rule

for the atomic reading and writing actions[13]:

$\texttt{x:}\ \sigma_1\ \texttt{(y:}\ \sigma_2\texttt{)} \rightarrow \sigma_1\ \texttt{= Channel(}\sigma_2\texttt{)}$

$\bar{x}\texttt{:}\ \sigma_1\ \langle\texttt{y:}\ \sigma_2\rangle \rightarrow \sigma_1\ \texttt{= Channel(}\sigma_2\texttt{)}$

In defining the delay π-calculus, I added *if* statements, delays and values. I therefore need to give these constructs typing rules. First, however, I need to define the type set.

In the system I have defined, names (i.e. variables) are either channels or values. Channels have the type rule refine above. Values can be of several types — Booleans, String, Characters and Numbers. I also allow a single name to stand for a tuple. There are two forms, fixed length tuples, and lists. Nothing else in the system is suitable for typing, therefore the set Σ of types for the language is:

$\Sigma\ \texttt{= \{Channel(}\sigma\texttt{), List(}\sigma\texttt{), Tuple (}\vec{\sigma_l}\texttt{), Bool, Num\}}$

Where l is the length of the tuple, and σ is a a member of Σ[14]. We can now define the rules for if statements, delays and value operators.

An if statement of the form *if* x *then* P *else* Q where P and Q are processes. The type rule for this is that x must have type Bool.

The numerical operators (plus, minus etc) have type Num and have the rule that the values which they are applied to also have type Num. The

[13]The actions themselves are not typed, but they force arguments they contain to be of a certain type

[14]Tuples were not added to the delay pi-calculus but, as described in section 2.2.1 they are a short-hand for simple name passing. The only extension here, is to allow single names to stand for tuples — which is done by the type-system

conditional operators, $(<, >, ==, <>)$, have type Bool, but the values they are applied to must have type Num.

The operations on lists are slightly more complicated. The *cons* operator, of the form $(x : xs)$ has the type "'List(σ)"', where the first argument (in the example x) has the type σ and the second argument has the type "'List(σ)"'. That is[15]:

`Cons x:` $\sigma_(1)$ `y:` $\sigma_2 \rightarrow \sigma_2$ `if` σ_2 `= List(`σ_1`)`

The *Head* operator applied to an argument has the type σ and it's argument has the type "'List(σ)"':

`Head x:` $\sigma_1 \rightarrow \sigma_2$ `if` σ_1 `= List(`σ_2`)`

The *Tail* operator has the output type "'List(σ)"' and input type "'List(σ)"':

`Tail x:` $\sigma_1 \rightarrow \sigma_1$ `if` σ_1 `= List(`σ`) for some` σ

Length returns a Num for input type Listσ, for arbitrary σ:

`Length x: List(`σ`)` $\rightarrow$ `Num`

Finally, *Concat* takes two lists of the same type and returns a list of the same type:

`Concat x: List(`σ`) y: List(`σ`)` $\rightarrow$ `List(`σ`)`

The tuple operations are easy. The tuple type contains a vector $\vec{\sigma}$ of the types at positions in the tuple. The tuple-element operator, therefore takes a tuple and returns the type in the position specified by the specific operator.

[15]somewhat confusingly, the symbol "':"' is taken to mean 'has type' in the following rule statements

4.3 Type Inference

Having defined the type-system, I move on to the system which will implement it. We want to have the benefits of a strong static typing, but without having to annotate everything with a type signature (although it is realized that there will be occasions where this is unavoidable). To achieve type inference, I will be using the Hindley-Milner system, and algorithm W. In the next section, I give a brief overview of algorithm W, and then discuss it in the context of the language we are designing.

4.3.1 Algorithm W

As discussed above (Section 4.2.6), we say that a process, or family of processes, is well-typed if it can be given a legal type assignment that is, all values in the process can be given at least one type (at least one to allow for polymorphic types). The role of type-checking then, is to try to find such an assignment. If it succeeds, we are guaranteed to avoid type errors in running the program[16].

The algorithm draws a distinction between *mono* types, or types with contain no other type information, and *poly* types, which do. This corresponds to what I termed *compound* types and *normal* types. In the interest of consistency, I shall stick to my original categorization.

The algorithm is based on unification[17]. Simply put, unification is the algorithm which takes a pair of expressions and either produces a substitution

[16](see Milner A Theory of Type Polymorphism in Programming) for a proof of soundness and a complete definition of the algorithm W

[17]See [19] for infomation on unification

which makes the two expressions 'equal' to each other (or unified) or it fails.

The algorithm can be expressed as a recursive function, and works as follows. The function has two arguments; the expression which is to be typed, and a partial function mapping expressions to types, called the environment. The return values of the function are the type of the expression, and a substitution required to make that expression that particular type. This substitution is a partial function which takes the type assigned to the expression by the environment to the type the expression is given by algorithm w. For each form an expression can take, rules are defined which detail type constraints on any of the sub expressions. For example, in an if expression of the form:

```
if e1 then e2 else e3
```

The type of $e1$ must be a boolean value (however the program deal with this) and the types of e2 and e3 must be the same. Ensuring these constraints are met is the domain of unification. Thus for the above if expression, algorithm W would work as follows (where τ is the function which maps an expression to a type):

τ`(if` e_1 `then` e_2 `else` e_3`)` $\rightarrow$ τe_2 `if` $\tau(e_1)$ `=bool and` $\tau(e_2)$ `=` $\tau(e_3)$

These rules are just the rules which we specified in Section 4.2.6. In the interest of brevity, I forgo restating these. This completes the design of the delay π-calculus. I shall now move on to the design of the higher-level language which allows elements of the functional paradigm to be translated into the underlying calculus.

4.4 Spiranda Syntax

Spiranda is a higher-level language which marries elements of functional programming with teh delay pi-calculus. It's semantics are given by it's translation down to this underlying language. The effect of Spiranda is to allow the programmer to choose between writing processes or functions[18] depending on which is more suited to the task in hand. The translation is Spiranda's key feature and therefore I will give this first.

4.4.1 Combining the best of both worlds

To simplify the expression of some algorithms, we want allow elements of to functional programming paradigm to be available. In particular, we want to include the following features:

- Higher-order functions — a function can either take a function as an argument, or return a function. This also give us curried function.
- Partial application of curried functions. A curried function should be able to be partially applied and then passed around. What is more, this partial application should be persistent — re-used in multiple applications.
- The functions should be referentially transparent. Enforcing this in the system has the nice property that it enforces the separation from the processing of information (which in the final system will be done with

[18] I mean functions in the functional programming sense — referentially transparent and allowing higher-order function

the functional syntax) with the storage of information (which will be done with processes)

With the above in mind, we can think of a function as a data-structure. Each function acts like a node which stores a bit of information and activates another node. The node activated will either be another function-node or it will be the operation node, which will compute the result of the function and return the result. These nodes can be made persistent by using replication. The following conversion I present is built on this idea.

The key idea in the conversion of a function in the pi-calculus is how to convert an application of an argument to a name. My approach is to model this as a dialog. The first action, is to sent a message to the *function* (i.e. the process which a function is translated into). This message will contain the first argument (or only argument) to the function, and a return name. The return name is a unique name created by the process carrying out the application, thus the return name provides the channel for responses. On receiving this ordered pair, the function starts another process. This second process will either be another function process or an operation process. The function process will be slightly different from the highest level function process as it has to create its own name (using restriction). Otherwise it functions in exactly the same way. The operation process, is responsible for computing and then returning the result.

The operation process likely includes function applications. This is the other side of the dialog. Applying an argument to a function is done as follows. Firstly, a return channel is created. Then this return channel and the argument are sent down the function channel (the name which is used to

communicate with the function). Finally the result is received on the return channel. If the result of the application is another function (i.e. the function is a curried function) we repeat the application process, but with the name received down the result channel as the name of our function. The use of replication means that this name can be re-used — it behaves just like a function at the highest level.

The above approach produces the following abstractions. The first is a function at the highest level[19]:

$$F(R) \;=\; (name, data)!(name(data2, return).((\nu \;\; sem)(\bar{sem}\langle \;(Cons \;\; data2 \;\; data)\rangle).0 \;\mid\; R \;\; < sem, return >))$$

Where R is an abstraction of arity two which is a parameter to F and is used to construct the result. It is activated by a semaphore called *sem* which passes the data along. The intermediate function process is defined as follows:

$$F'(R) \;=\; (sem, return)!(sem(data).(\nu \; n \, sem2)(\bar{return}\langle \;n\rangle.(F(R)\langle \;n \, data \;\rangle)))$$

However, if the following process is the operation of the function, then there will be an abstraction which takes the return name and the semaphore,

[19]The following processes would not be allowed in the pure pi calculus as I am sending a list down a channel and receiving it with a single name. However, in the delay p calculus you are allowed to do this — it is the binding of a name to value. The equivalent pure pi calculus term requires the use of another process (a list structure) to do this. This represents no additional difficulty, but I believe it is easier to follow as presented here.

and carries out any function applications which make up the operation of the function. An standard abstraction cannot be given for this function, but a procedure for creating it can be described.

The procedure will form an abstraction with arity two. The first value applied to the abstraction will be the semaphore which carries the data, and the second will be the function return channel. The procedure is as follows:

1. The first action will be to recieve the data along the semaphore channel.

2. If the function contains a *match* then this is converted and for each posibility, this procedure is carried out (from this step).

3. If the function contains a conditional construct, this is included next. The procedure is carried out from this point for the two function options in the *if* construct

4. A binding is created, for all of the name of intermediate values computed (i.e. the values which are inputs to other functions).

5. All of the function applications are carried out in parallel (the computation of a value is considered to be a function application, although it will require a different abstraction). These are ended by sending the final value out on the channel to which the result is bound. One of these function applications will return the result of the function.

As mentioned in the procedure above, the operation may consist of several function applications and these are carried out in parallel. This effectively recreates the action of *where* syntax seen in many functional languages. For example consider the S combinator:

```
S x y z =  x z (y z)
```

The right-hand side contains two two function applications. In a functional language like Haskell, this can be defined as:

```
S x y z = result
          where
          arg1 = y x
          result = x z arg1
```

Our conversion procedure is like this, as it specifies no order to have the function are computed (they are done concurrently). However, functions which take the result of other functions as arguments are actually evaluated first as the other functions will stall until the argument is available — i.e. until the first function has been evaluated.

The procedure outlined above relies on abstractions for carrying out function application, which are given next[20]:

$A(B)$ =
$(name, arguments, result)(\nu\, c)na\bar{m}e\langle(Head\ \ arguments), c\rangle.c(n2).$
$(B\langle\ \ n2, (Tail\, arguments), result\rangle)$

This is the application of a single argument to a function-process. In other words, it corresponds to the following:

```
name (Head argument)
```

[20] Again, the use of lists here is not required, but cleaner

Where *arguments* is the list of arguments to be applied in sequence. For example, the function application:

$$Fxyz$$

has the list of arguments $[x, y, z]$.

The abstraction B is the action which follows this application. This will be one of two things. If there is another argument to be applied then B will be another A abstraction. Otherwise, B will be an assignment process – a process to convert the result of the function application to a known value. This is the following:

$$A' = (name, arguments, result)\bar{result}\langle\ name\rangle.0$$

This just returns the value of the function application (which is passed to it as the *name* parameter for the abstraction) along the supplied result channel. This result channel can then be used as the input to another function, or as the result of the function (as outlined in the procedure given above).

Finally, I mentioned in the definition of the procedure for converting operations that the computation of a value was modeled as a function application. By this I meant that it was done in the same way as usual function applications — by its own process in parrallel with the function applcations. General abstractions for this cannot be given, as it would require operators to be passed to them which is not allowed, but it is easy to see how specific abstractions can be created. Consider the following process which adds two numbers which are passed along the channels 'arg1' and 'arg2':

```
Plus = (arg1,arg2,result)(arg1(x).arg2(y).result(x + y)
```

The inputs to this abstraction are different from the standard function application abstractions. We therefore define a conversion abstraction, so that the operator abstractions can be used in the same way as the function application abstraction.

$$C(Opp) \;=\; (name, args, result)(\nu\, p)(\bar{p}\langle(Head\ \ args)\rangle.\bar{p}\langle\ \ (Head\ \ (Tail\ \ args))\rangle.0$$
$$\mid\ \ p(arg1).p(arg2).(Opp\langle\ \ arg1, arg2, result\rangle))$$

Similar process abstractions can be defined for all of the operators in our extended pi calculus, as well as simple abstraction which return constant. This complete the translation of a function into the delay pi calculus. For clarification, examples of this conversion are given in Appendix H.

4.4.2 Functional Syntax

In designing the syntax for functional approach I use the functional language Miranda[21] [8] it is designed to mimic how mathematicians actually write functions as close as possible. Type-declarations, which can be applied to anything defined (functions and values are treated the same in the syntax) follow the Miranda style of using strings of asterisks, '$*$', for polymorphic type variables, and follow the same syntax of using ':' to mean *cons*, and '::' to mean *type*[22]. This gives us the following syntax:

```
<fdec>    ::= ‘fun’ <vars> ‘=’ <match>
<vars>    ::= <name> ‘ ’ <vars> | <name>
<match>   ::= ‘match’ <name> ‘with’ <cases> | <op>
```

[21] Miranda.org.uk

[22] In contrast with many functional languages, usually from the ML family

```
<cases>   ::= <case> ‘\n’ <cases> | <case>
<case>    ::= ‘case’ <pattern> ‘->’ <match>
<op>      ::= <if> | <exp>
<if>      ::= ‘if’ <value> ‘then’ <op> ‘else’ <op>
<exp>     ::= <expr> <cop> <exp> | <exp>
<cop>     ::= ‘:’
<expr>    ::= <term> <aop> <expr> | <term>
<aop>     ::= ‘+’ | ‘-’
<term>    ::= <comp> <muop> <term> | <comp>
<muop>    ::= ‘*’ | ‘/’
<comp>    ::= <bexpr> <comop> <comp> | <bexpr>
<comop>   ::= ‘<’ | ‘>’ | ‘==’ | ‘<>’
<bexpr>   ::= <factor> <fop> <bexpr> | <factor>
<factor>  ::= <paren> ‘ ’ <factor> | <paren>
<paren>   ::= ‘(‘ <op> ‘)’ | <atomarg>
<atomarg> ::= <name> | <literal>
```

A pattern is a limited form of a value, and is used for pattern matching. This I will define in Section 4.4.4.

4.4.3 Process Syntax

The process syntax is similar to the delay pi-calculus, but ammended to use charaters on standard keyboards. It also includes the new atom for getting a value from a function, which marries the function and process approaches.

```
<pdec>     ::= ‘proc’ <name> ‘with’ <values> ‘=’ <process>
```

```
<process>  ::= ‘replicate’ ‘(‘ <parallel> ‘)’ | <parallel>
<parallel> ::= <sum> ‘and’ <parallel> | <sum>
<sum>      ::= <action> ‘or’ <sum> | <action>
<action>   ::= ‘!’ <channel> ‘(‘ <value> ‘)’ ‘;’ <process>
             | ‘?’ <channel> ‘(‘ <value> ‘)’ ‘;’ <process>
             | ‘delay@’ <value> ‘;’ <process>
             | ‘[let’ <name> ‘=’ <application> ‘]’ <process>
             | ‘instantiate’ <name> ‘with’ <values>
             |  <if>
             | ‘(new’ <name> ‘)’ ‘(’ <process> ‘)’
             | ‘Zero’
             | ‘(’ <process> ‘)’
<if>       ::= "‘if"’ <value> "‘then"’ <process> "‘else"’ <process>
<literal>  ::= <bool> | <num> | <string> | <char>
<application> ::= <application> <value> | <value>
```

- The action starting with the bang, ! are the standard output actions and those starting with ? are the input actions
- The action *delay*@ is the newly added delay action
- The let binding allows a value to be bound to the return call of a function in a resulting process
- Replication is done with the keyword *replicate*
- Parallel composition of processes is done with the *and* keyword
- A summation is constructed with the *or* keyword

- Restriction is done with the new keyword, followed by the name, placed within parenthesis
- A abstraction is instantiated using the *instantiate* keyword. This is followed by a sequence of values which are the arguments to the abstraction.

The grammar enforces the following precedence of operations:

- Parallel composition is the weakest binding
- Summations are second
- Actions have the strongest binding and they associate to the right (the same as cons in functional programming)

For example, the following process:

```
? x(y); ! a(b); 0 | ? a(c); 0
```

Is equivalent to

```
(? x(y); ( ! a(b); 0 )) | (? a(c); 0)
```

If we wanted the action '? x(y)' to be followed by the parallel structure, we would have to write:

```
? x(y); ( ! a(b); 0 | ? a(c); 0 )
```

4.4.4 Pattern Matching

Pattern matching is introduced to the high level language. It is a simple way to express conditionals prevalent in functional programming. As such, in order for the high-level system to feel at all like a functional language, there must be a facility for pattern matching. Pattern matching lets the inputs to a function (or anything being assigned some value) be matched against values. This achieves two aims: specific values can be matched against a common example being matching against the empty-list in the base case of a recursive function, and different parts of a structure can be assigned different variables (or, indeed, specified to have specific values). As such, this structure can be encoded by a reduced collection of *allowable* values. These are all atomic values and the tuple and list structures. Other values, such as 'plus', are not allowed as the purpose of pattern matching is conditional branching (we only want simple values to be compared against).

Following is the pattern syntax, the collection of values which may comprise a pattern:

```
<pattern> ::= <list> | <tuple> | <name>
<list> ::= "‘(" ‘ <name> <cons> <xlist> "‘)"’
<xlist> ::= <name> <cons> <xlist> | <name>
<cons> ::= "‘:"’
<tuple> ::= "‘(" ‘ <innertuple> "‘)"’
<innertuple> ::= <value> "‘,"’ <xinnertuple>
<xinnertuple> ::= <value> "‘,"’ <xinnertuple> | <value>
```

To define multiple different *cases* for a particular function, the *match*

construct is used[23]. However, the delay π-calculus contains no such term. We therefore need a way of mapping the pattern-matching construct into the delay pi-calculus. This is not trivial, and I give this mapping next.

Translating Pattern-Matching into the Delay Π-Calculus

Pattern matching works by comparing the arguments supplied to a function against a selection of cases. If there is an assignment of the variables given in the case which can be given so that the case equals the same thing as the argument variables then this assignment is made, and the operation associated with that case is executed. If not, then the next case is evaluated. As such, a match statement does two things. Firstly, it assess the arguments, to see if the arguments can be unified with the particular case. Secondly, if the two can be unified, the assignments required to make the unification are made. This two step process can be converted to the delay π-calculus using the if construct, and actions which can bind names.

As above, the first step it to see if the case passes. This requires two types of checking: checking the item is the correct structure, and checking any values given in the case statement match. Structural checking is the more complicated.

For our purposes, as I have not included a facility for users defining their own types, structural checking will only include structural checking of lists. What this entails, is best seen by example.

Consider the following pattern for a list:

$$(x : xs)$$

[23]Similar to how SML based languages (Scala,F# etc) do pattern matching

This will match any list which can be split into a first element, *consed* onto a list. In other words, this matches any list which has more than one element. Similarly, the following pattern will match a list which has more than two elements:

$$(x : y : rest)$$

Thus structural matching of lists is just a conditional check that the length is greater than some minimum number. This is all we require for structural testing. Other structural failures, such as a tuple of incorrect length, represents type errors, so they needn't be checked against here.

Checking values in match cases is simple. It is simply the assertion that the variable being matched against is the same literal value as that given in the pattern. It is carried out after structural testing, so that parts of the structure can be matched against specific values. For example,$(x : 1 : xs)$ will be translated to a check that the length of the list is greater than or equal to two (greater than one). Then the second element (head of the tail of the original list) is checked against the value 1. If this passes, then the pattern is matched, and we move on to the assignment phase.

The assignment phase is carried out after a pattern has passed the structural and value tests and it allows the variables used in the match to be used in the operation associated with the match. Assignment is done entirely with pure π-calculus terms. The part of a structure which is assigned a variable is passed along a channel. This is read from the channel with a standard input atom. Input atoms bind the name they contain to the value they accept. Thus, if they are given the name of the variable to be bound, and they receive the value the name is to be bound to, we achieve a basic binding. That

is the following:

$$(\nu\ \mathtt{p})(\ \bar{p}\langle\ Head, x\rangle.0\ \mid\ \mathtt{p(y).Q})$$

Here the head of the list denoted by x is bound to the variable y in the process Q. Thus this can be used to bind variables to parts of structures. This completes the conversion from a *match* statement to delay π-calculus terms. Code which carries out this process can be found in Appendix F.3.

4.5 Design Summary

The design phase is complete. The language is based on an extended form of the pi calculus, which I have labeled the *delay π-calculus*. The language itself has features which allow the functional style of programming. This is just a shorthand, the functions are compiled down into the underlying calculus. Finally, the abstract machine for this calculus is beyond the scope of this project. Such a system would require a garbage collection system, as stalled processes need to be removed from memory[24]. Other strategies, such as a converter to another language to leverage that runtime engine are also possible. Further research is required before this can be achieved. However, a high-level description of a possible strategy for building the machine is given in Appendix G.

The analysis phase provided requirements for a suitable system for modeling the markets. Delays were added to the π-calculus to simplify the defi-

[24]Processes stalling is not due to bad program design, but is an unavoidable outcome of some processes

nition of a system with delays in. Arithmetic operators and data-structures were added to simplify task of doing arithmetic and building data structures which the analysis phase showed was difficult. Finally, a higher-level language was designed which allowed recurssive functions to be defined. The higher-level language is defined in terms of its translation into the delay pi-calculus. The effect is that the reduction semantics simply remains that of the delay π-calculus defined in Section 4.2.3.

Chapter 5

Implementation

5.1 Selecting the development language

The parser and type-system will be implemented using Haskell. This is attractive for two reasons. Its lazy functional semantics allow the use of Parser Combinators for building the parser[1]. A lot of the work in the program will also be based on manipulating the abstract, syntax tree. A functional approach, defining recursive functions over the tree-structure, seems perfect for this. We accept a potential loss in performance[2] for the easy of implementation. We can be equally cavalier about the space usage of Haskell[3] as this is still a way from a finished program, and performance is not yet an issue. As this is an ongoing development process, we choose the elegance of expression

[1]For more information about Parser Combinators, see [20] or [15] or [16]

[2]Although this is not necessarily the case: modern compilers of functional languages rival some *fast* languages for execution speed see http://shootout.alioth.debian.org/

[3]which remains one of the language's weak-points again, see http://shootout.alioth.debian.org/

over the speed of execution.

5.2 Building the Parser

As mentioned above, the ability to use parser combinators was a factor in choosing Haskell as the development language thus parser combinators are used to write the parser. I will give a brief description of what these are, before I go on to discuss why they are appropriate for the current project.

A parser can be thought of as a function from a string, the input text, to an abstract representation of the string the abstract syntax tree. As the parser in use may not consume the entire input string, we can amend this slightly, so that a parser returns the representation of the parsed sentence, and the unparsed input string. This gives us the following definition of the basic Parser type[16][4]:

```
newtype Parser a = Parser (String -> [(a,String)])
```

Parser combinators work with parsers on this principle. The combinators themselves are higher-order functions, which take specific parsers as input and return a parser as a result. The parser they return is the result of a particular grammar rule. For instance, in context free grammars (CFGs), it is common place for a production rule to have many options. For example:

```
Boolean ::= True | False
```

[4]Haskell syntax can be learned from http://www.haskell.org/haskellwiki/Haskell

For the above, we can define parsers for the tokens *True* and *False* and use a parser combinator to turn this into a parser for the *Boolean* non-terminal symbol. This can be defined in Haskell as follows:

```
(+++) :: Parser a -> Parser a -> Parser a
a +++ b = Parser (\cs -> case parse (a `mplus` b) cs of
                           [] -> []
                           (x:xs) -> [x])
```

where "`mplus`" is an associative operation defined as follows[5]:

```
mplus a b = Parser (\z -> ((parse a z) ++ (parse b z)))
```

The above parser conbinator brings the functionality of the '—' character in context-free grammars — it allows a non-terminal symbol to be defined as producing one or more option.

This is a trivial example, but similar parsers can be provided for all of the grammar rules - including recursive rules. This creates a system that is easy to build and, more importantly, easy to change.

This flexibility is ideal for the current project. With the use of the language, there may be new notations and language features which we would like to be able to include. Using parser combinators allows this to be done easily. It also allows the internal structure to be altered easily (such as expanding the type-system etc). There are drawbacks with the approach. Most significantly, they can use up to $O(exp)$ time and space. This is obviously not suitable for production systems, but for experimental systems this may

[5]It actually makes the Parser monad a member of the MonadPlus typeclass

be acceptable. However, care will have to be paid to the form of parser-combinators used, to streamline them as much as possible. This is covered in 'Monadic Parser Combinators' by Graham Hutton and Erik Meijer[16] the system I use will be made from the more-efficient combinators which they provide.

The representation which the parser creates also requires designing. The internal representation is the abstract syntax tree (AST) of the language we want to create. This can be represented easily by a Haskell algebraic data-type. For the process syntax we have the following:

```
data Process = Sum Process Process
             | End
             | Parallel Process Process
             | Restriction Binding Process
             | Instantiate ProcessName [(Value,Value)]
             | FunctionApplication Value Function Process
             | In (Variable,Value) Process
             | Out (Variable,Value) Process
             | Replicate Process]
             | Delay Value Process
             | If Value Process Process
             deriving (Eq,Show)
```

The simplicity with which this can be expressed shows one of the great advantages with Haskell for this task. I also chose to define a *Function* type as well. This is the type hich is returned when the parser parses a function.

This allows the parser to focus only on parsing, rather than having to parse and convert down to the process representation. The result of this is that after parsing, we require a processing step which converts these 'function' types into process types. This is a simple recursive function, which uses the same rule for converting functions as that given in Section 4.4.1. Although this function has to invent names for the conversion, there are no issues with binding, as these name are bound within the process being created, and their scope is internal to it.

The function abstract syntax tree is the following:

```
data Function = Assign Function Function
              | App Function Function
              | Val Value
              | Match Variable [Case]
              | Fif Value Function Function
            deriving (Eq,Show)
```

It should be noted that this is intended only as an intermediate structure. Rather than carry out the conversion while the parser is active, functions are parsed into a *Function* data structure and then this is converted into a process. This allows the functions to be written to perform one task — making the code easier to maintain or amend.

The most complicated part of the abstract syntax tree is the representation of *values* defined in Section 4.2.4. This contains all of the built-in operators, variables and literal values allowed[6]. This is as follows:

[6]The variables can be of any type so channels are held as variables

```
data Value =  --basic data typed
              Arg Variable --a variable or name
              | Lit Literal --a literal value
              --operations on numerical values
              | Plus Value Value
              | Minus Value Value
              | Times Value Value
              | Divide Value Value
              | Factor Value Value
              | Notequals Value Value
              | Equals Value Value
              | Lessthan Value Value
              | Greaterthan Value Value
              --operations on Boolean values
              | And Value Value
              | Or Value Value
              | Not Value
              --operations on lists
              | Cons Value Value
              | Head Value
              | Tail Value
              | Length Value
              | Concat Value Value
              --operations on tuples
              | Tuple [Value]
```

```
                | Tupleelement Value Value
                --used by parser if process includes function application
                | Fapplication Function
                --used as the end of a list/for empty messages
                | None
                deriving (Eq,Show)
```

That is the full abstract syntax tree. This is what the rest of the program creates and then manipulates.

5.3 Implementing Algorithm W

As outlined in Section 4.3.1, algorithm W can be expressed as a function which takes the environment, e, and the structure to be typed and returns the type of the structure together with a substitution of types required. In this section, I give the implementation issues which slightly altered how algorithm w was encoded. Full listing of the code can be found in Appendix F.2. The types form a tree structure which is encoded as follows:

```
data Type = TypeVar String
                  | Tupletype Int [Type]
                  | Listtype Type
                  | Channel Type
                  | Stringt
                  | Numt
                  | Chart
                  | Boolt
```

```
                  | Void
                  | Polymorphic
                  | FunctionType Type Type
  deriving (Eq)
```

The environment and the substitution can naturally be expressed as functions. The environment is a function from variable to types, and the substitution is a function from types to types. As the algorithm proceeds, these two functions will have to be 'updated'[7]. To allow a function to be updated, I used the following a higher-order function, which takes the existing environment, the new value and type to be added to the environment as arguments:

```
update :: (Variable -> Type) -> Variable -> Type -> Variable -> Type
update e x ty y = if x == y then ty else e y
```

When this is partially applied (to all but the last argument), we get the new environment. This environment has to be a total function (i.e. defined for all possible variable). To use this, I defined the empty environment (before any variables have been added to it) as:

```
emptyenv x = Polymorphic
```

This send all variable to the type 'Polymorphic' — indicating that they could be of any type. When this occurs, algorithm w assigns this variable a new type variable, updates the environment with this information and

[7]I use the term updated but in a functional language this is not literally true — data is immutable. Actually, the existing function is used as a building block in a new function (built by another function) which give the updated environment.

returns the type and no substitution (this is represented with the identity function — it makes no changes). As our version of algorithm w may change the environment, we need to add the environment to the return value[8].

The function which creates a unique type-variable also requires more information to be passed through the function w. In order to create a potential infinity of type variables, the function starts at the letter 'a', and proceeds upwards through the alphabet. In Haskell, functions are referentially transparent, so that the position of the last type-variable allocated must be given to the function in order for the next type variable to be unique. Furthermore Haskell does not allow the (easy) storage of state, so this variable must be stored by passing it into through each recursive call of algorithm w.

Finally, if the program is not well typed, algorithm w will fail. This is not a failure of the program, so we should not allow this failure to crash the system. Therefore, I have defined a monad[9] called 'Robust' which prevents this occurrence. In order to be used in a monad, the function return type must be a member of that monad. This gives us the following type for algorithm w[10]:

```
w :: ( (Variable -> Type),Int,a ) -> ...
```

[8]We could avoid this by using a function which makes a complete pass through the AST and build the environment first. However, it is both easier to code, and more efficient computationally, to do the two things at once

[9]A monad is, loosely, a computational context which allows the sequencing of operations. In this context, it checks the input of each function for an error marker. If there is one, it skips the subsequent actions. If no error is present, the function continues as usual. The code listing for this monad can be found in Appendix F.1

[10]Split over two line for formatting

```
... Robust ( (Type -> Type), (Variable -> Type), Int, Type )
```

Where 'a' is the structure to be typed. In our implementation, this structure is divided into multiple types (Haskell types — i.e. Processes, Values, Literals). In order to allow this function to be polymorphic over these types, I used a typeclass. This significantly tidies the implementation — otherwise we need multiple functions, each with different names, which act on different types.

This completes the implementation details of algorithm w. The algorithm itself was outlined in Section 4.3.1 and therefore will not be given again here. I now move on to consider the overall structure of the program[11].

5.4 Putting it all together

Finally, we have to put the components of the system in a specific order.

1. Parse Source
2. Check Scoping
3. Convert Functions to Processes
4. Type-check

Obviously, parsing is the first process. The structures created by the parsing process are then checked for the scoping of the names involved. Next, as parsing produces trees of type *function* (see Section 5.2), these are converted

[11]Selected code can be found in Appendix F

into process type. This is then type-checked. The type-checking occurs here as it requires the entire program to be checked together. Whilst the functions could be type-checked and then converted to processes, with the types discovered converted accordingly, I chose against this. It would add unnecessary complexity; there would have to be two implementations of algorithm W, one for processes and one for functions, and there would have to be two phases of type-checking. The potential drawback of converting first and then type-checking that it makes it more difficult to give useful error messages for type errors as the structure being checked is further away from the written structure. However, this is not so much of an issue in this early, developmental language.

5.5 Testing

Testing was carried out in stages. During the construction of the parser, when each component parser was finished, tests were written for it. These tests comprised of input strings. These were parsed and the abstract syntax tree produced was compared with the known correct representation. If any of these tests failed, then the parser was amended and checked against the test which fail. Once it passed, the whole testing phase for that parser was repeated (the changes made invalidated the results of the tests which had been parsed). These tests were kept and used to test all parsers which were constructed with the *or* parser combinator (see Section 5.2.

The parser was constructed first. For testing the type-inference system, the tests for the complete parser were used, along with others. The aim was

to give tests for all types of input program looking specifically at testing type-inference on recursive and polymorphic functions, as these represent the biggest difficulty for type inference systems. The function-to-process converter function was tested in the same way: function tree-structures with known process tree-structures were converted and then compared to the correct tree.

Each component of the progam was tested against multiple input values. Rather than having to apply each of these functions manually, I created a simple tester function. This carries out the function being tested for a given input, and then compares this with the correct value. If the test is passed, it returns true:

```
tester :: Eq b => (a -> b) -> a -> b -> Bool
tester f x y = if (f x) == y then True else False
```

This polymorphic function was used for all the testing carried out. I also defined a function which let a list of tests be applied repeatedly to the same function. The list contained tuples of input and output pairs, and it was defined as follows[12]:

```
testMany :: Eq b => (a -> b) -> [(a,b)] -> Int
testMany f list = xtestMany f list 0


xtestMany \_ [] \_ = -1
xtestMany f ((a,b):list) x = if (tester f a b) ...
```

[12]xtestMany will actually be a nested function of testMany, but they are written separately here for clarity

```
... then xtestMany f list (x+1) else x
```

This function returns -1 if all of the tests passed, else it returned the index of the test which failed. All testing was carried out within the Haskell interpreter a more convenient alternative to manually compiling and running all test programs.

Chapter 6

Summary and Conclusions

This project started with two aims: firstly to examine the suitability of existing modeling formalisms in modeling a financial markets with time delays; secondly, if appropriate, to extend these formalisms to give an improved system for modeling the markets.

The first aim was the subject of the analysis section, Section 3. This was the main body of the work. Three approaches were examined during this phase: a mathematic approach; the pure π-calculus and the stochastic pi calculus. While modeling the example hypothesis, it was found that the mathematical approach was more suitable for the description of the trader algorithms (Section 3.1.1). However, differential equations could not give a simple description of the interaction between the order-book and the traders. Conversely, the π-calculus was able to give a simple, concise, description of the interaction, but could not easily describe the trader algorithms involved. It also could not easily describe the delays in the system, which were a key part of our example hypothesis explaining the hot-potato effect (Sec-

tion 3.1.1). The stochastic π-calculus was rejected after examination, as the reduction semantics did not seem appropriate for our aims.

During phase two, I took the lessons from the analysis, and used them to create a new language which extended and combined the two approaches. A new *delay*, atom was added to the π-calculus to allow delays to be easily included in the model. A higher-level language was also defined, which let programs, or parts of programs, be written in a functional style. This allowed the algorithms involved in the market to be more easily modeled. The definition of this language was complemented with an implementation of a parser and converter, which took the high-level language parsed it, converted it down to the delay π-calculus and type-checked it.

Most of the contributions made was during the analysis phase. I assessed the suitability of the π-calculus, the Stochastic pi calculus and differential equations for modeling a financial markets. During this process, I created a way to do arithmetic in the pi calculus, and gave an implementation of a queue data-structure in the pure pi calculus. I also gave a method for filtering this data-structure when the members of the queue can be any name.

In the design phase, I extended the pi calculus to include arithmetic and basic data-structures. However, more importantly, I added a new *delay* construct to it, and created the reduction semantics for it. This makes it must easier to model systems with delays. I also gave a way of translating parts of the functional paradigm into the pi calculus. This allowed be to define a higher language, *Spiranda*, which allows the programmer to use functional syntax or process syntax — whichever is the best tool for the job. I finished with a step towards the implementation of the language, by creating a parser

and type-checker.

6.0.1 Further Work

There remains significant further work to be done:

- Further work is needed to expand the theory surrounding the delay π-calculus. Theories of bisimularity (when are processes the same) as needed, as well as soundness and completeness proofs.

- The language needs to be made executable. There are two possible strategies for this. An abstract machine could be created, possible along the lines of that outlined in Section G. This would also require a garbage collection system, as during execution many processes will be created which will then stall. These need to be recognized and then removed from memory. Alternatively, the delay *pi*-calculus produced by the existing system, could be converted to code in another language — taking advantage of existing garbage collection systems. If possible, this would save significant efforts. However, it is not trivial how any such translation would be done and the reduction semantics remain the same

- The language could be expanded and transformed into a simulator, along the lines of the Stochastic Pi Machine (http://research.microsoft.com/en-us/projects/spim/). This would require the language to be expanded to include directives to the simulator, detailing what needs to be plotted. Things which could be plotted are such as the values sent down a

specific channel, or the number of messages sent down a specific channel. This is a significant undertaking, but would provide a great tool for the formulation and exploration of hypothesis about the financial markets.

Appendix A

User Manual

The system is loaded in the way described in Appendix B. The system is very simple to use. It operates entirely from the command line. When run, it will prompt the user to enter the (relative) path of a file to compile. Once entered, it checks whether this file exists. If it doesn't, it displays an error and ends. Otherwise, it prompts the user to enter an output location. Once done, the system runs and exits. If no errors were encountered, the systems exits without any further message, returning to the command shell. Otherwise it displays the error and then quits.

The grammar for using the system is the Spiranda syntax, which is given in section 4.4. The file which contains it can be given any extension in windows (although .txt is usual).

Appendix B

System Manual

The code for the compiler can be found in a a folder on the DVD called 'Spiranda'. It contains a number of Haskell source files. In order to run these, they can either be compiled or run within the Glasgow Haskell Compiler's interactive environment.

Haskell can be downloaded from *http://hackage.haskell.org/platform/*. This is the standard Haskell platform, and include both the Glasgow Haskell Complier (GHC), the interpreter (GHCi) and the standard environment (prelude).

Once installed, the GHC can be used to compile the source. Copy the source from the DVD onto your hard-drive, keeping all the source-files in the same directory. From the command line, and in the directory the source is located in, type the following command:

```
ghc --make Main.hs
```

This will produce a file named Controller (or Controller.exe) which is the compiled program. This can be run from the command line.

To load the code into an interactive session, just load the Controller module (making sure all of the code is kept together). This can be done via the GUI in the windows version or from the command line. To load from the command line, load up GHCi by typing (assuming the Path is set up correctly):

```
ghci
```

and then change the directory to the directory in which the source code is located:

```
:cd path/to/correct/directory
```

Finally, type:

```
:load Main
```

Appendix C

Complete Addition Abstraction

This work represents my own approach to defining arithmetic within the π-calculus. It is based on Milner's *abstractions* [18] but the definition of all the terms is my own work.

From Section 3.3.3 we defined the following processes and abstractions:

The basic church numerals:

`0 := e`

`1 := me`

`2 := mme`

`...`

for which we have the following shorthand:

`n :=` $m^n e$

the addition process on church numerals

`Add(num1,num2,out,m,e) := (`ν `sem sem2)` $s\bar{e}m\langle\ \rangle$

$| \; !(sem().num1(x).\bar{x}\langle\rangle \;.0 \; | \; m().\bar{out}\langle \; m\rangle.s\bar{e}m\langle\rangle \;.0 \; \texttt{+} \; e().s\bar{e}m2\langle\rangle.0)$

$| \; !(sem2().num2(x).\bar{x}\langle\rangle \;.0 \; | \; m()\bar{out}\langle \; m\rangle.s\bar{e}m2\langle\rangle \;.0 \; \texttt{+} \; e().\bar{out}\langle \; e\rangle.0)$

the partially-complete integer abstractions

$I_0(m,e)(l) \;\; \texttt{:=} \; \bar{l}\langle \; e\rangle$

$I_1(m,e)(l) \;\; \texttt{:=} \; \bar{l}\langle \; m \;\; e\rangle$

$I_2(m,e)(l) \;\; \texttt{:=} \; \bar{l}\langle \; m \;\; m \;\; e\rangle$

$\ldots$

$I_n(m,e)(l) \;\; \texttt{:=} \; \bar{l}\langle \; m^n \;\; e\rangle$

the complete integer abstractions, of the form:

$N_n \;\; \texttt{:=} \; !(n(m,e,l).I_n\langle \; m,e\rangle\langle l\rangle)$

the addition abstraction which returns a church numeral

$\texttt{Addition(num1,num2,out) = (}\nu \; \texttt{m e n1 n2)} \; n\bar{u}m1\langle \; m \;\; e \;\; n1\rangle.n\bar{u}m2.\langle \; m \;\; e \;\; n2 \; \rangle.Add\langle \; n1,$

We need an abstraction which can take a numeral, store it, and act along the same lines as the Integer processes — taking input and then outputting the stored church numeral with the names given. As mentioned in the text, we need a queue for this. A queue is a first-in first-out storage data-structure. A queue was defined in Appendix D. We only need a simplified version of this:

$\texttt{Node(in,out,empty,sem) :=}$

$$(\nu\ \texttt{levelName})\ (sem(callingChannel).\bar{callingChannel}\langle\ levelName\rangle$$
$$.(in(dataItem).\bar{sem}\langle\ levelName\rangle\ .levelName(nextChannel).$$
$$callingChannel(anything).\bar{out}\langle\ dataItem\rangle\ .\bar{nextChannel}\langle\ nothing\rangle\ .0$$
$$+\ callingChannel(anything).\bar{empty}\langle\ nothing\rangle\ .0))$$

$$\texttt{Queue(in,out,empty) =}$$
$$(\nu\ \texttt{sem sem2 nothing end emptyx beginning})$$
$$(!(sem2(anything).(in(dataItem).\bar{sem}\langle\ beginning\rangle\ .beginning(nextChannel)$$
$$.\bar{out}\langle\ dataItem\rangle\ .\bar{nextChannel}\langle\ nothing\rangle\ .0$$
$$+\ \bar{empty}\langle\ nothing\rangle\ .empty(anything).\bar{sem}2\langle\ nothing\rangle\ .0))\ |$$
$$!(emptyx(anything).\bar{empty}\langle\ nothing\rangle\ .empty(anything).\bar{sem}2\langle\ nothing\rangle\ .0)$$
$$|\ !(Node\langle\ in\ out\ emptyx\ sem\rangle)$$
$$|\ \bar{sem}2\langle\ nothing\rangle\ .0)$$

This works by accepting input and placing the input on an output channel ready to be read. Simultaneously, it triggers a new process which accepts input and then triggers another process — ad infinitum. When the data is read from the output channel, a message is sent to the process which was triggered, prompting it to place its data on the output channel. When this item is read, the next Node process is triggered etc

We can therefore used this to store the cardinality of the number we want to capture. All that is left is to find a way of renaming the output to the names given on an input channel. We must also prevent any new data being added once we reach the empty signal. This process will have to be informed of the names in use first, before any of the names carrying the integer are sent to it. Putting all of this together, we get:

$$
\begin{aligned}
&\texttt{Result}(in_1\ in_2\ \texttt{o m e}) = (\nu\ \texttt{out i empty sem1 sem2}) \\
&\qquad\qquad \texttt{|}\ \bar{sem}1\langle\ \rangle.0 \\
&\qquad\qquad \texttt{|}\ !(sem1().in_1(x).\bar{x}\langle\ \rangle.0 \\
&\qquad\qquad\qquad \texttt{|}\ m().\bar{i}\langle\ m\ \rangle.\bar{sem}1\langle\ \rangle.0 \\
&\qquad\qquad\qquad \texttt{+}\ e().\bar{i}\langle\ e\ \rangle.0 \\
&\qquad\qquad \texttt{|}\ \texttt{Queue(i out empty)} \\
&\qquad\qquad \texttt{|}\ in_2(m_2\ e_2).(\bar{sem}2\langle\ \rangle.0 \\
&\qquad\qquad \texttt{|}\ !(sem2().out(x).x().0 \\
&\qquad\qquad \texttt{|}\ m().\bar{r}\langle\ m_1\rangle.\bar{sem}2\langle\ \rangle\ +\ e().\bar{r}\langle\ e_2\rangle))
\end{aligned}
$$

Finally, we can amend the addition process to return a name which accesses a process like that above:

$$
\begin{aligned}
&\texttt{Add2(num1,num2,out)} = (\nu\ \texttt{m e n1 n2 o result}) \\
&\qquad\qquad \bar{num}1\langle\ m\ e\ n1\rangle.\bar{num}2.\langle\ m\ e\ n2\ \rangle.Add\langle\ n1, n2, o, m, e\rangle \\
&\qquad\qquad \texttt{|}\ \texttt{Result(o result)}\ \texttt{|}\ \bar{out}\langle\ result\ \rangle
\end{aligned}
$$

We can output the name of the result process before the result has necessarily been calculated as we defined the Result abstraction is such a way that no action is possible on this channel until the result is complete.

This complete the definition of addition. It has the following characteristics

- numbers are encoded using church numerals
- numbers are stored in Number processes and accessed via unique names
- the values of m and e are bound when the numbers are accessed

- addition returns a *result* process which works in the same way as the Number processes

Appendix D

Implementing a Queue

In this appendix, I give an implementation of a queue in the π-calculus. A queue is an abstract data-structure with two basic operations; items can be added to the end of the queue and removed from the beginning. To make this task easier, we shall define an abstraction called *Node* which will store one data item. We'll then use this to create the *Queue* abstraction[1].

The Node abstraction must be able to receive a data item on an input channel (which we'll call *in*) and then put this onto an output (*out*) channel at the required time. As we don't want the length of the list to be limited, we will use replication to generate a new node. We therefore require a guard to prevent separate *Node* processes from interfering with each other. Taking these requirements into consideration, the *Node* can be defined as follows:

```
Node(in,out,empty,sem)=
```

[1] An abstraction is a process which contains unbound names. It can therefore act like a function, and be applied to different names in a given context - becoming a concretion (see section 2.3.4 or [18] for more information)

$(\nu$ `levelName`$)$ $(sem(callingChannel).callingChannel\langle\ levelName\rangle$
$.(in(dataItem).sem\langle\ levelName\rangle\ .levelName(nextChannel).$
$callingChannel(anything).out\langle\ dataItem\rangle\ .nextChannel\langle\ nothing\rangle$ `.0`
`+` $callingChannel(anything).empty\langle\ nothing\rangle\ .0))$

This works as follows:

1. A *Node* is activated by a message sent on the channel *sem*. This contains the name of the process which called it (*callingChannel*)

2. The *Node* returns it's name to the caller. This is needed so that the caller can inform this node that it has sent it's data, prompting this *Node* to output it's data.

3. (a) If the *Node* then receives data, it sends its name (*levelName*) out on the *sem* channel, starting a new *Node* process.

   (b) Finally, the *Node* receives a message from the *Node* which created it. This causes the *Node* to place it's data on the output channel. Once this has been read, it sends a message to the *Node* it activated, and dies.

4. (a) If the *Node* receives a message from it's caller before it receives any data, then the list is empty. It therefore sends out a message on the *empty* channel and dies.

This captures all of the behaviour we want from a node. A queue constructed of these components, will add new items at the end of the queue and remove from the beginning. It will also send the *empty* signal when the end is reached. We have been deliberately vague about what *data* is. Using

the shorthand of the polyadic π-calculus, this can be a vector of any length, so long as it is always the same length[2]

Now we can use the $Node$ abstraction to create a $Queue$ abstraction. This will have to use replication to allow for a potential infinity of nodes, as well as handle the cases where a current list dies because it is empty. Creating a context which implements these two factors, we get the following[3]:

```
Queue(in,out,empty) =
(ν sem sem2 nothing end emptyx beginning)
(!(sem2(anything).(in(dataItem).sēm⟨ beginning⟩ .beginning(nextChannel)
.ōut⟨ dataItem⟩ .nextCh̄annel⟨ nothing⟩ .0
 + em̄pty⟨ nothing⟩ .empty(anything).sēm2⟨ nothing⟩ .0)) |
!( emptyx(anything).em̄pty⟨ nothing⟩ .empty(anything).sēm2⟨ nothing⟩.0)
| !(Node⟨ in out emptyx sem ⟩)
| sēm2⟨ nothing⟩ .0)
```

The first replicative process (activated by $sem2$) acts as the first node – the same as the other nodes except it doesn't wait for a signal to output data. Thus if the first action executed is to receive data on in, it will activates a $Node$ and output the data. If the first action is a read, the process will output the *empty* signal and then return to the ready state. To prevent the chance of any interference, we have changed the output of the $Node$ empty signal

[2]Conceptually this is the same as a list of n-ary tuples in functional programming. Similarly, variable length data can be stored in each of the nodes in our queue, by constructing a queue of queues (or a list of lists etc).

[3]I use the notation of the abstraction name followed by braces containing the names it is applied to for concretions (abstractions bound to names)

to *xempty*. This is read by a process which then sends the empty signal, waits for confirmation on the same channel, and activates a new first-node, by outputting on $sem2$.

This completes the implementation of the queue. Although it looks complex, it is entirely self-contained. It has only three free names, *in*, *out* and *empty*. Data is stored by sending on *in* and retrieved by receiving on *out*. If there is no data stored, a signal will be sent on *empty* which must be confirmed by sending a message back on *empty*.

Appendix E

Removing a Limit Order by name

This appendix extends Appendix D with the functionality of filtering the elements of a queue. The difficulty here is that it requires a comparison where the domain of names which could potentially be part of the comparisson is infinite. This problem is addressed through the specific task of removing all the *limit orders* from a queue which have a specific name. In doing this, we will have to read the entire queue into a temporary queue, and then read it back to the storage queue, adding all items except the members of the limit order which we wish to cancel. This requires the definition of exactly what is stored as each data-item in the list.

Each limit order has four important bits of information connected to it:

- The unique name of the limit order
- The name of the trader who sent the order

- The value of the limit order
- The size of the limit order

The value of the limit order will define which tick process it is stored in, so we need not worry about it here. We are therefore left with storing the name of the limit order, the traders name and the size in the queue at each tick process[1]. We will store the size of the order in a similar way to that outlined in section ; based loosely on church numerals. However, rather than storing all the data items and then a end of data signal, we shall store the two together in a tuple – all of the data will be stored with a $moreData$ name, except the final item in the order, which will have a $lastItem$ signal. Although this may seem slightly contrived, it will simplify operations later on[2]. The other two bits of information we require, the trader name and limit order name, naturally form a pair. All the data can then be placed in a triple:

$$(X, Y, Z)$$

where $X = Ordername$, $Y = tradername$ and $Z = endOfOrder?$.

Having defined how the information is stored, we are now in a position to write how a item will be removed from the queue by order name. This will be done as follows:

[1]There will, of course, be two queues at each tick process – one for bids and one for asks. However, we need not worry about this here as the operation to remove a limit order will be the same for both queues.

[2]When executing market orders, we just count off single limit order items until the market order has bee filled. If we include end of data items in the list, we have to account for this whilst executing market orders - requiring us to add another comparison process to an already complex operation

1. (a) Read an item from the storage queue and send it back out on two different channels

   (b) Send one message containing the order name to a process which captures the name. Also send a copy a temporary queue for storage

   (c) Keep reading from the storage queue and adding into the temporary queue until the end of the order is reached.

   (d) If at the end of the queue, proceed to 2a, else go back to 1a

2. (a) Send a message out on the name of the order which we want to remove. It will send back a signal on $false$ so read this and discard. This completes the construction of the comparator - all the names are stored as processes which accept input on the order name, send a message back on $false$ and then die.

   (b) Create a process which receives on the name of the order we want to remove, and then sends this back out to a *trade cancellation confirmation* process

   (c) Create another process which receives a message on $false$, send one message out on the name of the limit order in the message and then sends the information contained in the message to the storage queue

   (d) Read from the temporary queue and output a message on the name of the limit order for the data item read

   (e) If the temporary queue is empty, then finish, else repeat 2d

To carry out step 1a, we require a process which received information on one channel, and then outputs it on two separate channels. We will call this a *Duplicator*:

$$
\begin{aligned}
&Duplicator\{in, out1, out2, killswitch\} = \\
&(\,\nu\, sem,\ nothing\,)\,(!(\,sem(anything).(killswitch(anything).0\ + \\
&in(x).\bar{out1}\langle x\rangle.\bar{out2}\langle x\rangle.\bar{sem}\langle nothing\rangle.0)\,)\ | \\
&\bar{sem}\langle nothing\rangle.0)
\end{aligned}
$$

The killswitch stops the process from replicating any more, as it does not send out on *sem* before it dies. This is included to prevent any interference between different stages in the algorithm. The process which captures the stored data and then responds to the name of the limit order we will call the *OrBuilder*. It is defined as follows

$$
\begin{aligned}
&(\,\nu\, sem,\ nothing\,)\,(!(\,sem(anything).(killSwitch(anything).0\ + \\
&in(orderName, traderName, isLastOrder).(\ (\nu\, sem2,\ sem3) \\
&((!(\,sem2(anything).in(orderName2, traderName2, isLastOrder2). \\
&\bar{isLastOrder2}\langle nothing\rangle.(sem3(anything).\bar{sem}\langle nothing\rangle. \\
&orderName(anything).\bar{out}\langle orderName\rangle.0\ | \\
&(notLastOrder(anything).\bar{sem2}\langle nothing\rangle.0\ + \\
&endOfOrder(anything).\bar{sem3}\langle nothing\rangle.0))\,))\ | \\
&\bar{sem2}\langle nothing\rangle.0)\ ))\,)\ |\ \bar{sem}\langle nothing\rangle.0)
\end{aligned}
$$

This seems complicated but it's actual operation is pretty simple. It is activated by receiving on *sem* and then either is killed with the *killswitch*,

which operates on the same lines as the killswitch for *Duplicate*, or it receives a triple on *in*. It then receives a triple again and, if it is that last item in the order, activates another process by sending a message on $sem3$. Otherwise it calls the inner process again – this repeats until the end of the order is reached. When a message is received on $sem3$, it activates another *OrBuilder* process and waits for input on the *orderName* channel. Once this has occurred, it sends *orderName* out on *out* and dies. In effect this builds a group of concurrent processes which all respond to a name in the list of orders. As they all respond on the same channel, they can be used as one side of a comparison process.

The *OrBuilder* only outputs each captured name once before dying. We therefore need a way of recreating a process which will response to an order name, as a single order will be spread across multiple items. We will achieve this using the process called *LightweightOrBuilder*. This works similarly to *OrBuilder*, but does not have to worry about storing more than one element with the same order name, as there will always be a one to one correspondence between reads and wries. This abstraction is easily defined as follows:

$$
\begin{aligned}
&LightweightOr\{in,\, out,\, killswitch\} = (\,\nu\, sem,\, nothing\,)\\
&(!(\,sem(anything).(killswitch(anything).0\; +\\
&in(name).s\bar{e}m\langle nothing\rangle.name(anything).o\bar{u}t\langle name\rangle.0)\,)\;|\\
&s\bar{e}m\langle nothing\rangle.0)
\end{aligned}
$$

This allows a name to be stored by sending it out on the channel *in*.

Similarly to the *OrBuilder*, each sub-process responds to a message on the channel of the name stored, sending the name stored out on *out*. If the output channel for this process is made to be the same name as the output channel for the *OrBuilder*, then the the behaviour is the same when a message is sent on the held name channel.

We now have to tackle the problem of comparing names. This is necessary as to remove all the item in a limit order, we need a way of deciding whether any given item is part of the limit order or not. We have already said that this will be done by comparing names, which is why we created the *OrBuilder* and *LightweightOr* abstractions – they give us a way of capturing all of the names which are currently in the queue. We will use a method similar to that outlined in section E to do the comparison. Our method for comparing the names will be as follows:

1. Read an item from the temporary queue if possible, else end
2. Send a message out on the limit order name channel
3. If the *limitOrderName* associated with the item just removed is the same as the name of the limit order we want to cancel, go to 4 else go to 5
4. (a) send the item details to a Handler process which will notify the trader that the item has been canceled
   (b) return back to 1
5. (a) send the item to the main queue for storage
   (b) if the item is the last in the limit order then go to 5d else go to 5c

(c) send the name of the limit order to the *LighweightOr* for storage and go back to 1

(d) send a message out on the *killswitch* for the *LightweightOr* and go back to 1

As the above shows, there are in fact two comparisons going on. The second is required so that by the end of the process, the *LightweightOr* has completely died, and thus will not interfere with any future procedures.

This is translated into the π-calculus as follows:

INSERT COMPARITOR ABSTRACTION

The handler abstraction, responsible for informing the trader that a limit order has been canceled, we will leave to be defined later. However we need to put some constraints on it for us to be able to full define the remove-by-name abstraction. From the outside, this abstraction will look pretty simple. There will be only one free name, the input channel, as the output channel will be the name of the trader whose order was canceled, and this information is contained in the information passed to it. Thus we will leave the Handler abstraction as follows:

$$Hander\langle in\rangle$$

Finally, we are left to defined the control logic of the process – the part of the abstraction which coordinates the whole procedure, activating and killing sub-processes as required. This was outlined in E, so all we are left with is how to translate this into the π-calculus. This is simple, and is done as follows:

```
ControlLogic(empty,orderName,tempout,false,handler,qin,
```

activateDuplicator,oKill,dKill,cKill,tempempty,
processingFinished,cKill,lwin,lwkillswitch,lwOrActivator)
:=
(ν nothing)$activate\bar{D}uplicator(nothing).empty(anything).$
$em\bar{p}ty(nothing).d\bar{K}ill(nothing).o\bar{K}ill(nothing).$
$lwOr\bar{A}ctivator(nothing).order\bar{N}ame(nothing).false(any).$
$((tempempty(a).c\bar{K}ill(nothing).processing\bar{F}inished(nothing).0$
| $Comparator\langle tempout, orderName, false, qin, cKill, lwin, lwkillswitch\rangle))$

All of this can be put together, to give the complete process for removing a limit order by name:

RemoveOrder(orderName qout qin empty endOfOrder handlerout processingFinished)
=
(ν tempin tempout tempempty oKill lwkill dkill cKill
true false handler cKill lwin lwkillswitch lwOrActivator
activateDuplicator obin)
lwOrActivator(any).
$LightweightOr\langle lwin, false, lwkillswitch\rangle$ |
$Handler\langle handler\rangle$
$activateDuplicator(anything).Duplicator\langle qout, tempin, obin, dkill\rangle$ |
$LightwieghtOr\langle\ lwin, false, lwkill\rangle$ |
$OrBuilder\langle obin, false, endOfOrder, oKill\rangle$ |
$Queue\langle\ tempin, tempout, tempempty\rangle$ |
$ControlLogic\langle$empty orderName tempout false handler
qin activateDuplicator oKill dKill cKill tempempty
processingFinished lwin lwkillswitch lwOrActivator$\rangle$

This just implements what was outlined in the algorithms given above. The is a significant proliferation of names, which represents one of the problems of working in the π-calculus. This is due to the number of guards and semaphores required to stop separate processes from interfering with each other. However, this added complexity is only superficial. At it's core, the above process remains very simple.

Appendix F

Selected Code Listing

During this section I use the notation ... to end a line and start a line to show that the line should wrap around.

F.1 Robust Monad

```
module Robust where
--Robust is an error tolerant wrapper for any data type.
-- It contains two constructors, one for the non-error
-- version and one which includes a string error message
data Robust a = Fine a | Error String
              deriving (Eq)


instance Show a => Show (Robust a) where
   show (Fine a) = show a
   show (Error s) = "Error " ++ s


instance Functor Robust where
```

```
    fmap f (Fine a)  = (Fine (f a))
    fmap f (Error s) = (Error s)



instance Monad Robust where
    (>>=) (Error s) _ = Error s
    (>>=) (Fine a)  f = f a
    return a = Fine a


foldRobust :: (a -> (Robust b) -> (Robust b))...
  ... -> (Robust b) -> [a] -> (Robust b)
--foldRobust f def [] = def
--foldRobust f def (x:xs) = do { y <- (foldRobust f def xs); f x y }
foldRobust = foldr


isError :: Robust a -> Bool
isError (Error s) = True
isError _       = False


--simple deconstructor - removes the Robust wrapper
extractRobust :: Robust a -> a
extractRobust (Fine a) = a
extractRobust _        = error "Tried to remove a wrapped error"


--other deconstructor
extractError :: Robust a -> String
extractError (Error s) = s
extractError _         = error "error from non-error Robust"


--joins two Robust list,  concaternating the
```

```
concatRobust :: (Robust [a]) -> (Robust [a]) -> (Robust [a])
concatRobust (Fine a)  x = fmap (a ++ ) x
concatRobust (Error s) x = Error s
```

F.2 Type Inference

```
module TypeInference2(w,annotate,emptyenv) where

import AbstractSyntaxTree
import Typesystem2
import Robust
import Char

identity :: x -> x
identity = id

emptyenv x = Polymorphic

--function to create  new type var, essential
-- part of algorithm w
newTypeVar :: Int -> Type
newTypeVar c = TypeVar ([chr (97 + c)])

--typeclass to allow Variables/Literals/Processes/Values be treated the same
class Typeable a where
     w :: ( (Variable -> Type),Int,a ) -> ...
        ...Robust ( (Type -> Type), (Variable -> Type), Int, Type )
     annotate :: (Variable -> Type) -> a -> a
```

```
--used to ensure the scope: variable scope at highestlevel
--in the ProcessWrapper is maintained
instance Typeable a => Typeable [a] where
     w (env,i,[]) = Fine (identity,env,i,Void)
     w (env,i,(x:xs)) = do (s1,e1,i1,t1) <- w (env,i,x)
                           (s2,e2,i2,t2) <- w (e1,i1,xs)
                           return ((s2.s1),e2,i2,Void)
     annotate s x = map (annotate s) x

--Introduces new type variables if variable is not bound in scope,
-- or else produces the typevar this variable has been assigned
instance Typeable Variable where
     w (env,i,(Typedvar s t)) = Fine (identity,env,i,t)
     w (env, i,var) | env var == Polymorphic = let ty = newTypeVar i in
                                                  Fine (identity,(update env var ty),(i+1),ty)
                    | otherwise = Fine (identity,env,i,(env var))
     annotate env (Variable s) = Typedvar s (env (Variable s))
     annotate env x = x

instance Typeable Literal where --returns the type corresponding to the literal given
     w (env, i,lit) = Fine (identity,env,i,(getLitType lit))
     annotate e l = l

instance Typeable CompleteProcess where --needs a process dictionary, a tree of names and
     w (env,i,(CompleteProcess (n,b,p))) = w (env,i,p)
     annotate e (CompleteProcess (n,b,p)) = CompleteProcess (n,(annotateBinding e b),(annotate e p))

--IMPORTANT HERE TO GET SCOPING RULES CORRECT
-- the updated envionment/etc is only used when THE VARAIBLE SCOPE CONTINUES
instance Typeable Process where
```

```
    annotate e p = case p of
                    (Sum p1 p2) -> Sum (annotate e p1) (annotate e p2)
                    (Parallel p1 p2) -> Parallel (annotate e p1) (annotate e p2)
                    (In (var,val) p1) -> In ((annotate e var),(annotate e val)) (annotate e p1)
                    (Out (var,val) p1) -> Out ((annotate e var),(annotate e val)) (annotate e p1)
                    (Delay val p) -> Delay val (annotate e p)
                    (End) -> End
                    (Replicate p) -> Replicate (annotate e p)
                    (Restriction b p) -> Restriction (annotateBinding e b) (annotate e p)
                    (If v p1 p2) -> If (annotate e v) (annotate e p1) (annotate e p2)
    w (env, i,p) = case p of
--the types of all processess are Void, but values are typed.
--Here, w just makes sure that all of the typings are carried out
                    (Instantiate n tlist) -> foldl foldFunction (Fine (id,env,i,Void)) tlist
                                             where
                                             foldFunction = \x1 y -> x1 >>= (\(s1,e1,i1,t1) ...
                    ...->  do {(s2,e2,i2,t2) <- w (e1,i1,y); return  ((s2.s1),e2,i2,Void)})
                    (Sum p1 p2) -> do (s1,e1,i1,t1) <- w (env, i,p1)
                                      (s2,e2,i2,t2) <- w (env, i1,p2)
                                      -- the scope of p1 does NOT extend to p2
                                      return ((s2.s1),env,i2,Void)
                    (Parallel p1 p2) -> do (s1,e1,i1,ti) <- w (env, i,p1)
                                           (s2,e2,i2,t2) <- w (env, i1,p2)
                                           -- Again, the scope of p1 does NOT extend to p2
                                           return ((s2.s1),env,i2,Void)
                    (In (var,val) p1) -> do (s1,e1,i1,t1) <- w (env, i,var)
                                            (s2,e2,i2,t2) <- w (e1, i1,val)
                                            --the scope DOES extend
                                            s3 <- unify (t1,(Channel t2))
                                            return ((s3.s2.s1),e2,i2,Void)
```

```
                    (Out (var,val) p1) -> do (s1,e1,i1,t1) <- w (env, i,var)
                                             --the scope DOES extend
                                             (s2,e2,i2,t2) <- w (e1, i1,val)
                                             s3 <- unify (t1,(Channel t2))
                                             return ((s3.s2.s1),e2,i2,Void)
                    (Delay val p) -> w (env,i,p) --scope extends, but nothing added
                    (End) -> Fine (identity,env,i,Void)
                    (Replicate p) -> w (env,i,p) -- scope extends, but nothing added
                    (Restriction b p) -> w (env,i,p) -- scope extends, nothing added
                    (If v p1 p2) -> do (s1,e1,i1,t1) <- w (env,i,v)
                                       s2 <- unify ((s1 t1),Boolt)
                                       (s3,e2,i2,t2) <- w (env, i,p1)
                                       (s4,e3,i3,t3) <- w (env, i1,p2)
                                        -- the scope of p1 does NOT extend to p2
                                       return ((s4.s3.s2.s1),env,i2,Void)

-- no new binding constructs can introduce names into values, so no need to worry about scope here
instance Typeable Value where
     annotate e val = case val of
                        (Arg v) -> Arg (annotate e v)
                        (Lit l) -> Lit l
                        (Plus v1 v2) -> Plus (annotate e v1) (annotate e v2)
                        (Minus v1 v2) -> Minus (annotate e v1) (annotate e v2)
                        (Times v1 v2) -> Times (annotate e v1) (annotate e v2)
                        (Divide v1 v2) -> Divide (annotate e v1) (annotate e v2)
                        (Factor v1 v2) -> Factor (annotate e v1) (annotate e v2)
                        (Equals v1 v2) -> Equals (annotate e v1) (annotate e v2)
                        (Notequals v1 v2) -> Notequals (annotate e v1) (annotate e v2)
                        (And v1 v2) -> And (annotate e v1) (annotate e v2)
                        (Or v1 v2) -> Or (annotate e v1) (annotate e v2)
```

```
                    (Concat v1 v2) -> Concat (annotate e v1) (annotate e v2)
                    (Lessthan v1 v2) -> Lessthan (annotate e v1) (annotate e v2)
                    (Greaterthan v1 v2) -> Lessthan(annotate e v1) (annotate e v2)
                    (Not v) -> Not (annotate e v)
                    (Tuple vlist) -> Tuple (map (annotate e) vlist)
                    (Tupleelement index tuple) -> Tupleelement index (annotate e tuple)
                    (Cons v1 v2) -> Cons (annotate e v1) (annotate e v2)
                    (Head v) -> Head (annotate e v)
                    (Tail v) -> Tail (annotate e v)
                    None -> None
w (env, i,val) = case val of
 (Arg v) -> w (env, i,v)
 (Lit l) -> w (env, i,l)
 (Plus v1 v2) -> numTwoTuple (env, i,(v1,v2))
 (Minus v1 v2) -> numTwoTuple (env, i,(v1,v2))
 (Times v1 v2) -> numTwoTuple (env, i,(v1,v2))
 (Divide v1 v2) -> numTwoTuple (env, i,(v1,v2))
 (Factor v1 v2) -> numTwoTuple (env, i,(v1,v2))
 (Equals v1 v2) -> do (s1,e1,i1,t1) <- w (env, i,v1)
                      (s2,e2,i2,t2) <- w (e1, i1,v2)
                      s3 <- unify (((s2.s1) t2),((s2.s1) t1))
                      return ((s3.s2.s1),e2,i2,((s3.s2.s1) t1))
 (Notequals v1 v2) -> do (s1,e1,i1,t1) <- w (env, i,v1)
                         (s2,e2,i2,t2) <- w (e1, i1,v2)
                         s3 <- unify (((s2.s1) t2),((s2.s1) t1))
                         return ((s3.s2.s1),e2,i2,((s3.s2.s1) t1))
 (And v1 v2) -> binaryOperator Boolt (env, i,(v1,v2))
 (Or v1 v2) -> binaryOperator Boolt (env, i,(v1,v2))
 (Concat v1 v2) -> binaryOperator Stringt (env, i,(v1,v2))
 (Lessthan v1 v2) -> do{ (a,b,c,d) <- numTwoTuple (env, i,(v1,v2)); return (a,b,c,Boolt)}
```

```
        (Greaterthan v1 v2) -> do { (a,b,c,d) <- numTwoTuple (env, i,(v1,v2)); return (a,b,c,Boolt)}
        (Not v) -> do (s1,e1,i1,t1) <- w (env,i,v)
                      s3 <- unify (((s1) t1),Boolt)
                      return ((s3.s1),e1,i1,Boolt)
        (Tuple vlist) -> foldl foldFunction (Fine (identity,env,i,(Tupletype 0 []))) vlist
                         where
                         foldFunction = \x1 y -> x1 >>= (\(s1,e1,i1,(Tupletype a list)) -> ...
         ... do {(s2,e2,i2,t1) <- w (e1,i1,y); return ((s2.s1),e2,i2,(Tupletype (a+1) (t1:list))) })
        (Tupleelement index (Tuple tuple)) -> do (s1,e1,i1,t1) <- w (env,i,index)
                                                 s2 <- unify (t1,Numt)
                                                 (s3,e2,i2,t2) <- w (e1,i1,tuple)
                                                 s4 <- unify (t2,(makeTupleType i1 (length tuple)))
                                                 return ((s4.s3.s2.s1),e2,(i2 + (length tuple)),((s4.s3) t2))
        (Cons v1 v2) -> do (s1,e1,i1,t1) <- w (env, i,v1)
                           (s2,e2,i2,t2) <- w (env, i,v2)
                           s3 <- unify (((s2.s1) t2),(Listtype ((s2.s1) t1)))
                           return ((s3.s2.s1),e2,i2,((s3.s2.s1) t2))
        (Head v) -> do (s1,e1,i1,t1) <- w (env,i,v)
                       s2 <- unify ((Listtype (newTypeVar i1)),(t1))
                       return ((s2.s1),e1,(i1 + 1),(s2 (newTypeVar i1)))
        (Tail v) -> do (s1,e1,i1,t1) <- w (env,i,v)
                       s2 <- unify ((Listtype (newTypeVar i1)),(t1))
                       return ((s2.s1),e1,(i1 + 1),((s2.s1) t1))
         None -> Fine (identity,env,(i+1),(newTypeVar i))
         -- None is polymorphic (like the empty list) so give it a new typevar
         _ -> Error "Ill-formed AST"
--converts a binding to a binding with annotated variables
annotateBinding :: (Variable -> Type) -> Binding -> Binding
annotateBinding e (Binding l) = Binding (map (annotate e) l)
```

```
makeTupleType :: Int -> Int -> Type
makeTupleType i l = Tupletype l (makeList i l)
                    where
                    makeList i 0 = []
                    makeList i l = (newTypeVar i) : (makeList (i+1) (l-1))

numTwoTuple :: ( (Variable -> Type),Int,(Value,Value)) -> ...
     ... Robust ((Type -> Type), (Variable -> Type), Int, Type)
numTwoTuple = binaryOperator Numt

binaryOperator :: Type -> ( (Variable -> Type),Int,(Value,Value)) -> ...
     ...Robust ((Type -> Type), (Variable -> Type), Int, Type)
binaryOperator t (env, i,(v1,v2)) = do (s1,e1,i1,t1) <- w (env, i,v1)
                                       s2 <- unify ((s1 t1),t)
                                       (s3,e2,i2,t2) <- w (e1, i1,v2)
                                       s4 <- unify (((s3.s2.s1) t2),t)
                                       return ((s4.s3.s2.s1),e2,i2,t)


--updates the environment (when a new variable is bound to a type variable)
update :: (Variable -> Type) -> Variable -> Type -> Variable -> Type
update e x ty y = if x == y then ty else e y

--replace is used as the function which maps type to type in the
-- main function w (replaces type variables with concrete types)
replace :: (Type,Type) -> Type -> Type
-- the first of the tuple is the type var to be replaced by the second
replace (a,t) (Tupletype l list) = Tupletype l (map (replace (a,t)) list)
replace (a,t) (Listtype x) = Listtype (replace (a,t) x)
replace (a,t) (Channel x) = Channel (replace (a,t) x)
```

```
replace (a,t) x = if a == x then t else a


--Queries whether the first type (composite or type variable) is found in the second type
occurs :: (Type,Type) -> Bool
--occurs (s1,(FunctionType x y)) = if (occurs (s1,x)) then True else occurs (s1,y)
occurs (s1,(Tupletype l list)) = (dropWhile (== False) (map (\a -> (occurs (s1,a))) list)) /= []
occurs (s1,(Listtype x)) = occurs (s1,x)
occurs (s1,(Channel x)) = occurs (s1,x)
occurs (x,y) = if x == y then True else False


--the unification function, returns the assumption require
--to make the two types input equal to each other
unify :: (Type,Type) -> Robust (Type -> Type)
unify ((TypeVar b),(TypeVar a)) = if a == b then Fine identity else Fine $ replace ((TypeVar b),(TypeVar a))
unify ((Channel a),(Channel b)) = unify (a,b)
unify ((Listtype a),(Listtype b)) = unify (a,b)
unify ((Tupletype l1 a),(Tupletype l2 b)) | l1 /= l2 = Error $ "Unable to unify tuples of different size"
                                          | otherwise = ...
...foldr (\(x,y) z1 -> ( z1 >>= ( \z -> do { s <- unify ((z x),(z y)); return (s.z)}))) (Fine identity) (zip a b)
unify ((TypeVar a),t) | (TypeVar a) == t           = Fine identity
                      | occurs ((TypeVar a),t)     = Error $ "circularity error"'
                      | otherwise                  = Fine (replace ((TypeVar a),t))
unify (a,(TypeVar b)) = unify ((TypeVar b),a)
unify (x,y) = if x == y then Fine identity else Error $ "type mismatch in types: "
```

F.3 Function to Process Converter Function

```
module FunctionConverter3(convertParseWrapper) where


import AbstractSyntaxTree
```

```
import Robust
import ASTManipulation


convertParseWrapper :: ParseWrapper -> Robust ProcessWrapper
convertParseWrapper (a,b,c) = do convertedFunctions <- convertFunctions a
                                 convertedProcesses <- convertProcesses b
                                 return (convertedFunctions ++ convertedProcesses)


--functions to carry out the conversions over lists
convertFunctions :: [Function] -> Robust [CompleteProcess]
convertFunctions [] = Fine []
convertFunctions (x:xs) = do {f <- convertFunction x; fs <- convertFunctions xs; return (f:fs)}


--function carries out the conversion over a list
convertProcesses :: [CompleteProcess] -> Robust [CompleteProcess]
convertProcesses [] = Fine []
convertProcesses (x:xs) = do {f <- convertCompleteProcess x; ...
 ...fs <- convertProcesses xs; return (f:fs)}


--the abstractions required to make the converstion
--abstractions are modeled by a function from a tuple to a process
--if the abstraction is parametric on a process,
-- this is included before the the tuple (allowing partial application)


fF :: ((Value,Value) -> Process) -> (Value,Value) -> Process
fF rR (name,dat) = Replicate (In (name,(Tuple [(Arg dat2),(Arg return)])) ...
...(Restriction (Binding [sem]) (Parallel (Out ((Arg sem),(Cons (Arg dat2) dat)) End) ...
...(rR ((Arg sem),(Arg return))))))
                 where
                 dat2 = Variable "data2"
```

```
                return = Variable "return"
                sem = Variable "sem"

--the Fprime abstraction - for anonymous functions (i.e. the functions returned during currying
fFprime :: ((Value,Value) -> Process) -> (Value,Value) -> Process
fFprime rR (sem,return) = Replicate (In (sem,dat) (Restriction (Binding [name2]) ...
   ...(Out (return,(Arg name2)) (fF rR ((Arg name2),dat)))))
                    where
                    name2 = Variable "name2"
                    dat = Arg (Variable "data")

--the application abstraction.  This has been slightly ammended Now the args are channels through which the
--arguments for the function application can be recieved (same as the arg for the value abstractions)
aA :: ((Value,Value,Value) -> Process) -> (Value,Value,Value) -> Process
aA bB (name,args,result) = Restriction (Binding [c]) (In ((Head args),value) ...
  ...(Out (name,(Tuple [value,(Arg c)])) (In (Arg c,(Arg n2)) (bB ((Arg n2),(Tail args),result) ) )))
                     where
                     c = Variable "c"
                     n2 = Variable "n2"
                     value = Arg (Variable "value")

--the final application abstraction, returns the result of the
--function application along the channel required (result)
aAprime :: (Value,Value,Value) -> Process
aAprime (name,args,result) = Out (result,name) End

binOpAbstraction :: (Value -> Value -> Value) -> (Value,Value,Value) -> Process
binOpAbstraction op (arg1,arg2,result) = In (arg1,x) (In (arg2,y) (Out (result,(op x y)) End))
                                       where
                                       x = Arg (Variable "x")
```

```
                                    y = Arg (Variable "y")

--all the following abstractions have the type: (Variable,Value,Variable) -> Process
plus = binOpAbstraction Plus
minus = binOpAbstraction Minus
times = binOpAbstraction Times
divide = binOpAbstraction Divide
factor = binOpAbstraction Factor
notequals = binOpAbstraction Notequals
equals = binOpAbstraction Equals
lessthan = binOpAbstraction Lessthan
greaterthan = binOpAbstraction Greaterthan
xand = binOpAbstraction And
xor = binOpAbstraction Or
cons = binOpAbstraction Cons
xconcat = binOpAbstraction Concat
tupleelement = binOpAbstraction Tupleelement


--the unary operations require no conversion
unOpAbstraction :: (Value -> Value) -> (Value,Value,Value) -> Process
unOpAbstraction op (dummy,args,result) = Out (result,(op (Head args))) End

xnot = unOpAbstraction Not
xhead = unOpAbstraction Head
xtail = unOpAbstraction Tail
xlength = unOpAbstraction Length


--now the functions which use the above abstractions to define the process-functions
```

```
convertFunction :: Function -> Robust CompleteProcess
convertFunction (Assign a1 a2) = do (abstraction,args) <- convertFunctionValue a1
                                    p <- convertOperation (a2,args)
                                    return (CompleteProcess ("",(Binding []),abstraction p))

--converts the left-hxand-side of a function definition to a process
-- - the action of taking arguments xand returning partial functions
--the list is return as well, so the operation builder can identify
--the correct position in the list for each of the funciton arguments
convertFunctionValue :: Function -> Robust ((((Value,Value) -> Process) -> Process),[Value])
convertFunctionValue (App (Val (Arg v1)) (Val a2)) = Fine ((\x -> fF x ((Arg v1),None)),[a2])
--the first abstraction, parametric on a second abstraction, x
convertFunctionValue (App al (Val a2)) = do {(a,b) <- convertFunctionValue al; ...
 ...return ((\x -> a (fFprime x)),(a2:b))}

--convert operation converts the right-hxand side of the
--function definition.  It is split into different levels
--convert operation returns an abstractions
-- with the required input - variable variable
--importantly this abstraction is NOT parametric on any other abstraction
convertOperation :: (Function,[Value]) -> Robust ((Value,Value) -> Process)
convertOperation (f,args) = Fine (\(sem,return) -> In (sem,dat) ...
 ...(stageTwo (f,args,return))) --args is the contents of dat
                           where
                           dat = Arg (Variable "dat")

stageTwo :: (Function,[Value],Value) -> Process
stageTwo (f,values,return) = case f of
   (Match v cases) -> matchConversion (f,values,return)
   _ -> stageThree (f,values,return)
```

```
stageThree :: (Function,[Value],Value) -> Process
stageThree (f,values,return) = case f of
   (Fif v f1 f2) -> If v (stageThree (f1,values,return)) (stageThree (f2,values,return))
   _ -> let (p,i) = (topStageFour (f,values,return,0)) in p


--the top level.  It applies the final abstraction, producing the required function.
--- This is concurrent with the other processes accrued along the way
topStageFour :: (Function,[Value],Value,Int) -> (Process,Int)
topStageFour (f,values,return,i) = case f of
  (App a1 a2) -> let newArg = makeNewArgument i
                     (process,int1) = topStageFour (a2,values,(Arg newArg),(i+1))
                     (abst,int2) = xstageFour (a1,values,(Cons (Arg newArg) None),return,int1)
                 in ((Parallel (abst aAprime) process),int2)
  (Val v) -> stageFourValue (v,values,return,i)


--the is not the top level - i.e. this has treversed down the spine of a function.
-- It therefore returns something parametric on an abstraction
xstageFour :: (Function,[Value],Value,Value,Int) -> ...
 ...((((Value,Value,Value) -> Process) -> Process),Int)
xstageFour (f,values,args,return,i) = case f of
  (Val v) -> let newArg = makeNewArgument i
                 (p,i2) = stageFourValue (v,values,(Arg newArg),(i+1))
             in (\x -> Parallel p (aA x (v,args,return)),i2)
             --then this is the name of the function - treat as such xand continue
  (App a1 a2) -> let newArg = makeNewArgument i
                     (process,int1) = topStageFour (a2,values,(Arg newArg),(i+1))
                     (abst,int2) = xstageFour (a1,values,(Cons (Arg newArg) args),return,int1)
                 in  ((\x -> (Parallel (abst (aA x)) process)),int2)
```

```
--This converts the values to suitable processes for
== inclusion (the must be processes to get the scoping correct)
stageFourValue :: (Value,[Value],Value,Int) -> (Process,Int)
stageFourValue (v,values,return,i) = case v of
  (Arg v) -> if elem (Arg v) values then ...
   ...((Out (return,(getArgFromData (Arg v) values)) End),i)else ((Out (return,(Arg v)) End),i)
  (Lit l) -> ((Out (return,(Lit l)) End),i)
  (Plus v1 v2) -> bOpConvert plus i v1 v2 return values
  (Minus v1 v2) -> bOpConvert minus i v1 v2 return values
  (Times v1 v2) -> bOpConvert times i v1 v2 return values
  (Divide v1 v2) -> bOpConvert divide i v1 v2 return values
  (Factor v1 v2) -> bOpConvert factor i v1 v2 return values
  (Equals v1 v2) -> bOpConvert equals i v1 v2 return values
  (Lessthan v1 v2) -> bOpConvert lessthan i v1 v2 return values
  (Greaterthan v1 v2) -> bOpConvert greaterthan i v1 v2 return values
  (And v1 v2) -> bOpConvert xand i v1 v2 return values
  (Or v1 v2) -> bOpConvert xor i v1 v2 return values
  (Cons v1 v2) -> bOpConvert cons i v1 v2 return values
  (Concat v1 v2) -> bOpConvert xconcat i v1 v2 return values
  (Tupleelement v1 v2) -> bOpConvert tupleelement i v1 v2 return values
  (Head v) -> unOpConvert xhead i v return values
  (Not v) -> unOpConvert xnot i v return values
  (Tail v) -> unOpConvert xtail i v return values
  (Length v) -> unOpConvert xlength i v return values
  (Fapplication f) -> topStageFour (f,values,return,i)
  (Tuple t) -> tupleValue ((Tuple t),values,return,i)
  (None) -> (End,i)

getArgFromData :: Value -> [Value] -> Value
getArgFromData v [] = error "should never get here"
```

```
getArgFromData v (x:xs) = if (v == x) then Head (Arg (Variable "dat")) else Tail (getArgFromData v xs)


--simplified the process for all of the repetitive binary operators
bOpConvert :: ((Value,Value,Value) -> Process) -> Int -> Value -> Value -> Value -> [Value] -> (Process,Int)
bOpConvert abs i v1 v2 return values = let newArg1 = Arg (makeNewArgument i)
                                           newArg2 = Arg (makeNewArgument (i+1))
                                           (p1,i1) = stageFourValue (v1,values,newArg1,(i+2))
                                           (p2,i2) = stageFourValue (v2,values,newArg2,(i1))
                                           plusprocess = abs (newArg1,newArg2,return)
                                       in  ((Parallel p1 (Parallel p2 plusprocess)),i2)


--simplifies the process for all unary operators (same process by with different abstractions)
unOpConvert :: ((Value,Value,Value) -> Process) -> Int -> Value -> Value -> [Value] -> (Process,Int)
unOpConvert abs i v1 return values = let newArg1 = Arg (makeNewArgument i)
                                         (p1,i1) = stageFourValue (v1,values,newArg1,(i+1))
                                         newprocess = abs ((Arg (Variable "dummy")),newArg1,return)
                                     in ((Parallel p1 newprocess),i1)


--special case for a tuple - more complicated so defined here
tupleValue :: (Value,[Value],Value,Int) -> (Process,Int)
tupleValue (t,values,return,i) = let (p2,i2) = xtupleValue (t,values,i,End)
                                     newProcess = constructTupleProcess i i2 return
                                 in ((Parallel p2 newProcess),i2)


xtupleValue :: (Value,[Value],Int,Process) -> (Process,Int)
xtupleValue (t,values,i,p) = case t of
   (Tuple []) -> (p,i)
   (Tuple (x:xs)) -> let newArg = Arg (makeNewArgument i)
                         (p1,i1) = stageFourValue (x,values,newArg,(i+1))
                     in xtupleValue ((Tuple xs),values,i1,(Parallel p p1))
```

```
--construcTupleProcess binds all of the names in the tuple into a single tuple - to be sent along return

constructTupleProcess :: Int -> Int -> Value -> Process

constructTupleProcess i1 i2 return = xconstruct i1 i2 id [] return


xconstruct int1 int2 p1 v2 ret | int1 == int2 = p1 (Out (return,(Tuple v2)) End)

                               | otherwise    = let np1 = \x -> p1 (In (Arg ...

                                  ...(makeNewArgument int1),(makeTupleArg int1)) x)

                                                    nv2 = v2 ++ [makeTupleArg int1]

                                               in xconstruct (int1 + 1) int2 np1 nv2 ret


makeNewArgument :: Int -> Variable

makeNewArgument i = Variable ("__arg" ++ (show i))


makeTupleArg :: Int -> Value

makeTupleArg i = Arg (Variable ("tupleArg" ++ (show i)))


--matchConversion converts a 'match' construct to the underlying calculus

matchConversion :: (Function,[Value],Value) -> Process

matchConversion ((Match var []),vlist,vars) = End

matchConversion ((Match var (x:xs)),vlist,vars) = If (structureTests (Arg var) x) ...

 ... (matchProcess var x (vlist,var)) (matchConversion ((Match var xs),vlist,vars))


--matchProcess will create a process will carry out the necessary assignments

--(from the matching conditions) xand then execute the process converted from the function

--which is the second part of the case expression (i.e. the function to be carried out if the match conditions are

--

matchProcess :: Variable -> Case -> ([Value],Variable)-> Process

matchProcess vl1 (Case (vl2,v)) (vlist,var) = (assign (Arg vl1) vl2) ...

 ...(stageTwo (v,vlist,(Arg var)))
```

```
--assignments carries out the name assignments which are made by the match
-- e.g. match x case (y:ys) assigns y to head x xand ys to tail x.
--these must be carried out BEFORE the process can execute.  Hence this
-- function returns a function Process -> Process which
-- sequences these actions before the function result
assignments :: [Value] -> [Value] -> Process -> Process
assignments [] [] = id
assignments (x:xs) (y:ys) = \z -> (assign x y) (assignments xs ys z)

--assign assigns the varaibles in the pattern y to the constituent parts of x
--x will start as a variable, but operations will be added to it on recurssions
assign :: Value -> Value -> Process -> Process
assign x (Cons v1 v2) = \z -> (assign (Head x) v1) (assign (Tail x) v2 z)
assign x (Tuple (v:vs)) = tupleAssign 0 x (v:vs)
                        where
                        tupleAssign i x [] = id
                        tupleAssign i x (y:ys) = \z -> (assign ...
    ...(Tupleelement (Lit (Nlit (fromIntegral i))) x) y) ((tupleAssign (i+1) x ys) z)
assign x (Arg v) = let newname = Variable "~private"
                   in  \z -> Restriction (Binding [newname]) ...
    ...(Parallel (Out ((Arg newname),x) End) (In ((Arg newname),(Arg v)) z))
assign x (Lit l) = id --in this case, as it is a match, the thing
--being matched against will already have this value.  Therefore we can ignore


--structure tests converts the part of the match conversion checking
-- that the variables have the correct structure.  This will be carried out on lists
--as we only allow matching on lists xand tuples xand tuples
--cannot be of different length if the function is properly typed
```

```
--it will also be carried out on literal values
structureTests :: Value -> Case -> Value
structureTests x (Case (y,f)) | isList y    = (Greaterthan (Length x) ...
 ...(getLength y)) --pattern matching against lists
                               | isLiteral y = (Equals x y)
                               --pattern matching against specific literals
                               | otherwise   = (Lit (Blit True))

--function isList sees if the Value encoded is a list structure
isList :: Value -> Bool
isList (Cons v1 v2) = True
isList _ = False

--function isLiteral sees if the Value is a literal value
isLiteral :: Value -> Bool
isLiteral (Lit l) = True
isLiteral _ = False

--function getLength returns the Value containing the
-- length of the encoded pattern lit minimum length
getLength :: Value -> Value
getLength = toValue.xgetlength
            where
            xgetlength (Cons v1 v2) = 1 + (xgetlength v2)
            xgetlength _ = 0
            toValue x = Lit (Nlit (fromIntegral x))

--Replaces all function application terms in the
..complete process with the correct pi-calculus process
convertCompleteProcess :: CompleteProcess -> Robust CompleteProcess
```

```
convertCompleteProcess (CompleteProcess (a,b,c)) =  ...
      ...Fine (CompleteProcess (a,b,(convertProcess c)))


--A process can call functions to get values.  This converts down into delay pi caclulus terms
convertProcess :: Process -> Process
convertProcess (FunctionApplication v f p) = Parallel (stageTwo (f,[],v)) p
convertProcess (Sum p1 p2) = Sum (convertProcess p1) (convertProcess p2)
convertProcess (Parallel p1 p2) = Parallel (convertProcess p1) (convertProcess p2)
convertProcess (Restriction b p) = Restriction b (convertProcess p)
convertProcess (In t p) = In t (convertProcess p)
convertProcess (Out t p) = Out t (convertProcess p)
convertProcess (Replicate p) = Replicate (convertProcess p)
convertProcess (Delay v p) = Delay v (convertProcess p)
convertProcess x = x
```

Appendix G

Outline of an Abstract Machine for the Delay Π-Calculus

As previously stated in Section 3.5, it is outside the scope of this project to implement an abstract machine for this calculus. However, I give a brief outline of how this could be achieved.

The program will be held in a tree data-structure. This tree will contain branching structures, such as parallel composition '—' and summations '+', and non-branching structures: replication '!', restriction and the standard in and out actions. The abstract machine will work in two phases; exploration followed by reduction.

In the exploration phase, the tree will be traversed and all the actions in unguarded position copied into a table, along with a pointer to the position in the tree. Once all accessible actions have been visited[1] and entered into the table, the exploration phase is complete, and the reduction phase begins.

[1] Accessible here means in unguarded position — i.e. see footnote on previous page

The table will in fact be a table of tables. Each name which is acting as a channel (rather than as a message) will be given its own table[2]. This will have a column for reading actions and one for writing actions. In this column, the content of the message will be stored along with the pointer to the action. With this table, we can calculate the number of possible different reductions for one channel by multiplying the number of entries in one of these columns by the number of entries in the other.

Delays will be stored in a different table, as a link list sorted by length. Rather than update the entire list when each delay reduction is carried out, we can just increment a counter in the program which will hold the current *time*. When added to this list, the delay will be stored with the value of the current time added to the length of this particular delay. When a delay is *reduced*, the delay is removed from the list, and the counted changed to the value of the delay which was reduced.

The reduction phase starts with a check for possible standard reductions. If there are any possible standard reductions, one of these will be picked at random (each with an equal probability of being picked), the tree updated and action will return to the exploration phase. If there is not a standard reduction, then the shortest delay will be reduced, the tree and delay environment updated, and the program would return to exploration.

The exploration phase, in practice, will be short for all phases following the first reduction. This is because the tree will remain the same except for the reduction which has just occurred. This will be the only part of the tree that needs to be explored. Reductions which occur in summations will also

[2]Two identical names with different scopes will have different tables

require the removal of actions from the actions table, as all of the options except one will be removed from the tree.

This complete the outline of a simple abstract machine.

Appendix H

Example Conversions of Functions into Delay Π-Calculus Terms

H.0.1 The *factorial* function

Factorial function in a functional language:

```
fun factorial x = if x == 1 then 1 else (factoral (x-1))*x
```

And after our conversion:

$F(F'((sem, return)(sem(data).$
$(if\ ((Head\ data)\ =\ 1)\ then\ (P_1\langle\ date, return\rangle)\ else(P_2\langle\ data, return\rangle))))))$

where

$P_1\ =\ (data, return)\bar{return}\langle\ 1\rangle$

$$P_2 \;=\; (data, return)(\nu\ arg1)((A(A')\langle\ factorial, data, arg1\rangle) \mid (arg1(x).return((Head\ data)*(x))))$$

The above definition is split to make it easier to follow — it is of no importance.

H.0.2 The I combinator

The combinator I is defined functionally as follows:

```
fun I x = x
```

This simple function, can be defined in terms of the

$$F((sem, return)(sem(data).\bar{return}\langle\ Head\ data\rangle))\langle\ I, []\rangle$$

H.0.3 The K combinator

The K combinator is defined as follows:

```
K x y = x
```

And in the pi calculus:

$$F(F'((sem, return)(sem(data).\bar{return}\langle\ (Head(Tail\ data))\rangle)))\langle\ K, []\rangle$$

H.0.4 The S combinator

$F(F'(F'$
$(sem, return).(sem(data).$
$(\nu\, arg1)(A(A_\prime)\langle\ ((Head\ (Tail\ data)), (Head\ (Tail\ (Tail\ data))), arg1\ \rangle \quad \mid$
$A(A(A_\prime))\langle\ (Head\ data), (Cons\ (Cons\ (Head\ (Tail\ (Tail\ data)))\ arg1)\ Nil), result\rangle))))$

Where $Head\,(Tail\,data))$ returns the name y, $Head(Tail(Taildata))$ returns the name z and $Headdata$ returns the name x. Whilst this is far from pretty, the conversion is easily computed. The upshot of this is that a functional syntax can be used, and then the above abstractions used to convert it down into the π-calculus, allowing elements of the functional paradigm to be incorporated into our modeling language.

Appendix I

Selected Tests and results

Following are the tests for the parser. When run in the Haskell interpereter, they run without error (they test specific component parts of the system)

```
--function definitions

parse arguments "a b c d e f g"

parse arguments "'a' b"

parse arguments "a \"b\" c d e f g"

parse arguments "a \"b\" 0000 0.12345"

--pattern parsing

parse listpattern "a:as"
```

```
parse listpattern "a:b:c"

parse listpattern "a:as"

parse listpattern "'a':as"

parse listpattern "'a' : as"

parse listpattern "(a:as):rest"

parse pattern "((a:as):rest)"

parse pattern "(True:as)"

parse pattern "((True):as)"

parse pattern "((Constructor a 'a'):as)"

--arguments

parse factor "12 ^ 12"

parse expr "2 + 10"

parse expr "2*10 + 2"

parse expr "2*10 + 2 / (2 + 3) + 1"

parse operation "f (1*2) 3 'a' (ff x y z)"
```

```
parse partialFunction "f x = x"

parse partialFunction "f x = 1"

parse partialFunction "f x = 1 + 2"

parse partialFunction "contains s t = not (isError (get s t))"

parse partialFunction "f x = ((1 + 2))"
```

Bibliography

[1] Samadi Mehrdad Andrei A. Kirilenko, Albert S. Kyle and Tuzun Tugkan. The flash crash: The impact of high frequency trading on an electronic market, May 2011.

[2] James J. Angel. Tick size, share prices, and stock splits. *The Journal of Finance*, 52(2):pp. 655–681, 1997.

[3] Marco Avellaneda and Sasha Stoikov. High-frequency trading in a limit order book, 2006.

[4] Deutsche Bank. High frequency trading: better than its reputation?, 2011.

[5] J. A. Bergstra. *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA, 2001.

[6] Luca Cardelli and Radu Mardare. Stochastic pi-calculus revisited.

[7] C. Clack, C. Myers, and E. Poon. *Programming with Miranda*. Prentice Hall (New York), 1995.

[8] US Commodity Futures Trading Commission, US Securities, and Exchange Commission. Findings regarding the market events of may 6, 2010, 2010.

[9] Easley D., M. Lpez de Prado, and M. O'Hara. The microstructure of the 'flash crash': Flow toxicity, liquidity crashes and the probability of informed trading". *The Journal of Portfolio Management*, 37:118128, 2011.

[10] Willem F. Duisenberg. The role of financial markets for economic growth, 2001.

[11] Thierry Foucault, Ohad Kadan, and Eugene Kandel. Limit order book as a market for liquidity. *Review of Financial Studies*, 18(4):1171–1217, Winter 2005.

[12] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[13] Fabien Guilbaud and Huyen Pham. Optimal high frequency trading with limit and market orders, 2011.

[14] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators an Introduction*. Combridge University Press, Cambridge, UK, 2008.

[15] Graham Hutton. Higher-order functions for parsing, 1993.

[16] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[17] Nanex LLc. Nanex flash crash summary report, 2010.

[18] Robin Milner. *Communicating and mobile systems: the pi-calculus*. Combridge University Press, Cambridge, UK, 1999.

[19] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[20] S. Doaitse Swierstra. Combinator parsers: From toys to tools, 2001.