A domain-specific language for agent-based modelling in Haskell

TIM COWLISHAW
Department of Computer Science
University College London
Malet Place
London WC1E 6BT, UK
t.cowlishaw@cs.ucl.ac.uk

September 7, 2011

This report is submitted as part requirement for the MSc Computer Science degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text.

Abstract

This report introduces a framework for agent-based modelling in the functional programming language Haskell (Peyton Jones 2003).

Agent-based modelling (Holland & Miller 1991) is a strategy for computational modelling based on simulating the interactions of multiple autonomous agents, and observing the behaviour emerging from these interactions. We argue that functional languages such as Haskell are particularly well suited to agent-based modelling and simulation tasks, and that Haskell in particular offers several advantageous features.

Our modelling framework is implemented as an embedded domain-specific language, and is biased towards applications of ABM that require the semantics regarding timing of events to be well specified and deterministic, for instance where the analysis of the results is founded in systems engineering techniques, as in (Clack 2011).

We also discuss the rationale for both these design choices and present a review of the state of the art in both agent-based modelling and DSL design. Finally, we present a demonstration of the use of our framework in the form of a case study investigating the effect of variable attenuation of trading rates on liquidity and volatility in a simple securities market, based on the work presented in (Clack 2011).

Contents

1 Introduction

Agent-based approaches to modelling and simulation provide a novel approach to the study of complex systems which concentrates on describing simulations as systems of multiple interacting agents in which emergent properties of the entire system can be observed, in contrast to more traditional approaches to mathematical modelling which attempt to fully specify the behaviour of a model through a system of equations relating observable properties of the system as a whole (Van Dyke Parunak, Savit & Riolo 1998). Such an approach can have many advantages- it allows researchers to build upon knowledge of the behaviour of individuals within a system to achieve knowledge of the behaviour of the system as a whole (rather than relying on knowledge of the macro-level behaviour of the system to create a model in the first place), and allows the modelling of emergent effects that would not otherwise have been observed.

Agent-based approaches to modelling and simulation have been used with success in many fields, to model a diverse range of subjects, including the effect of a reduction in tick size on a financial market (Darley & Outkin 2007) and levels of burglary in a UK city (Malleson, Heppenstall & See 2010). However, despite the wide ranging applications of this technique, many tools for producing agent-based simulations require a high level of programming ability on the part of the researcher conducting the simulation, or access to a programmer who can implement the model.

We argue that an approach to defining agent-based simulations based upon a system of interacting Domain-specific languages (Ghosh 2011) mitigates this problem, providing an interface for describing simulations which is expressive enough to model a wide variety of different simulations and behaviours, but which is simple enough for researchers with little experience programming in a general-purpose language to interact with effectively. We define such a system, with a particular focus on simulating the interactions between traders on a single-security limit-order market (based on the work in (Clack 2011)), providing a case-study simulating the effects of dynamically attenuating trading rates on levels of price volatility and liquidity within the market. Our system should allow the definition of simulations in a high level, declarative manner, using the language of the domain being modelled, and should interface with existing tools for the analysis and presentation of results. We argue that functional programming languages, and in particular the strongly-typed, lazy functional language Haskell (Peyton Jones 2003) provide a suitable environment for our framework, and present the use of a number of common functional programming idioms and techniques which offer particular advantages for our purposes.

2 Background

2.1 Agent-based modelling

Agent based approaches to simulation and modelling were first proposed in (Holland & Miller 1991), and are based on the science of Complex Systems, networks of heterogenous, interacting agents which display emergent properties resulting from the interaction of these agents, and which are not directly observable from the actions of any single agent. As outlined in the introduction, this approach offers advantages in modelling systems where the overall behaviour is not directly observable. In particular, it has been used to model financial markets with a great deal of success, by accurately modelling the behaviour of individual economic actors, we can use ABM to observe properties of the overall market that would otherwise be difficult to model. For instance, (Clack 2011) takes this approach to identify the existence of feedback loops and oscillations within a financial market emerging from the interactions of the market participants - identifying such patterns through the use of a top-down, equational approach to modelling would be difficult, due to the emergent nature of these properties. In addition, ABMs have been used to predict the consequences of a change in tick size on the NASDAQ exchange (Darley & Outkin 2007). As a result of the global economic downturn, and the flash crash of May the 6th 2010, there is currently a huge amount of interest in the use of novel approaches (and especially Agent-based modelling) to broaden our understanding of financial markets and improve regulation (Farmer & Foley 2009).

Many frameworks for producing agent-based computational models exist, two of the most popular being MASON (Luke, Cioffi-Revilla, Panait, Sullivan & Balan 2005) and Swarm (Minar, Burkhart, Langton & Askenazi 1996), which support a huge variety of different simulation topologies and types, and offer integrated features for visualising simulation results. Both of these are implemented in the Java programming language, and are heavily reliant on an understanding of object-oriented modelling techniques, and the Java language, on the part of the researcher implementing the simulation. We propose that a different approach to the design of a simulation framework might afford greater ease of use on the part of the researchers implementing the simulation, more specifically, by defining a smaller, more specific toolset which interfaces with existing tools to provide features such as statistical analysis and visualisation, we can provide an interface which is more readily usable by a wider variety of people.

2.2 Domain-specific languages

Both our simulation framework and the case study are designed as embedded domain-specific languages (eDSLs). A domain-specific language denotes a specialist programming language which expresses entities, relationships and operations in the same vocabulary as the domain being modelled (Ghosh 2011). This approach has the benefit of allowing increased clarity in communication with specialists in the domain to be modelled (Ghosh 2011). This ensures that requirements can be easily gathered, from which software can be created whose application logic can then be verified by domain experts who may not be familiar with general-purpose programming languages. DSL-based development also affords a more expressive programming style, using declarative constructs that map intuitively to concepts in the domain to be modelled (Van Deursen, Klint & Visser 2000). In addition, A DSL-based development approach is very amenable to the use of formal methods for verifying program correctness. (Hudak 1996).

Embedded (or '*internal*') domain-specific languages are DSLs which are expressed within a *host* language, allowing the developer of the DSL to build upon the existing capabilities of the language, and the user of the DSL to combine domain-specific constructs with the generic features of the host where necessary (Ghosh 2010). *eDSLs* also allow multiple DSLs implemented within the same host language to be composed (Mak 2007), augmenting their expressive power, especially in applications with functionality whose concerns lie in the intersection of the domains modelled by one or more DSLs. By contrast, *external* DSLs are implemented as standalone languages, with their own compiler or interpreter and runtime environment. This can allow greater expressiveness, as the DSLs syntax is not constrained by that of a host language. However, external DSLs are more costly to develop (due to the increased work involved in writing a language runtime from scratch), and lack the benefit of composability afforded by an embedded DSL.

For these reasons, we chose to implement our simulation framework as an embedded, rather than external DSL. The Haskell language offers many features that are advantageous for DSL-based development, and for dealing with specific properties of the domains of simulation and financial modelling. In particular, it has a very concise syntax which also affords a large amount of flexibility to the DSL designer. Function application is denoted by juxtaposition of terms and is left-associative, unlike java or C-style languages which require function calls to be postfixed with parentheses, with the order of application made explicit; Haskell also allows functions to be called with their arguments in either infix or prefix position. All these features greatly improve readability of code written in a domain-specific language. Its expressive type system allows much of the logic of Haskell programs to be encoded at the type level, allowing the compiler to detect a great many bugs that would otherwise lead to runtime errors.

This project concerns itself with several domains of discourse, each of which we model individually as a domain-specific language. In particular, we define a DSL for describing agent-based simulations in the abstract, as well as another for describing securities orders in a limit-order book. These two languages are then composed within our case study simulation, allowing us to express the behaviour of traders in our market simulation in a high-level, declerative manner.

The process of designing a DSL necessarily relies on a solid understanding of the domain to be modelled, coupled with a sound design mapping this understanding into software. (Ghosh 2010) describes this process as creating a mapping from the *problem domain* being modelled to the *solution domain*; the domain of tools, methodologies and techniques in which our domain specific language will be expressed.

This mapping process involves identifying the key entities in the problem domain at an appropriate level of granularity, and ensuring that they are each represented by a corresponding entity in the solution domain, such that all interations and relationships between entities are preserved by the mapping. The process of creating this mapping involves the creation of a vocabulary common to both the problem and solution domains, a technique grounded in Eric Evans' concept of a *ubiquitous language*, a key part of the methodology of *Domain-driven design* (Evans 2003). In order to do this, the language used to describe entities and relationships in the problem domain as the vocabulary used by the domain-specific language. Given that our solution domain is a strongly typed, functional programming language, these building-blocks of our domain specific language will be composed of types, typeclasses and combinators, following the method described in (Peyton Jones & Eber 2003).

2.3 Volatility and liquidity in financial markets

Our case-study is based heavily on the work presented in (Clack 2011), which examines the events of May 6th 2010 from a systems engineering perspective, examining the effects of feedback loops and oscillations caused by the interactions between traders on volatility and liquidity in a limit-order book. There is a wealth of additional information and analysis already published about the 'flash crash', in particular, (Kirilenko, Kyle, Samadi & Tuzun 2010) provides a taxonomy of traders based on data recording trades in the Standard & Poors E-mini 500 futures contract on May the 6th, along with descriptions of the trading strategies employed by each category of trader. We use this information to inform the behaviour of the trading agents within our simulation.

3 Analysis & Design

The field of agent-based modelling describes a simulation in terms of *interactions* between heterogenous, independent *agents*, which have *behaviours* which influence their interactions. (Macal & North 2010). Furthermore, as outlined in (Clack 2011), our agents should interact through a sequence of shared *states*, such that the agents' interactions with the state are ordered in a well-defined and deterministic manner. As a result, we can infer that our DSL must include types representing the following entities in the problem domain: *Simulation*, *Behaviour*, *Agent State* and *Interaction*. (Clack 2011) sets out the semantics of how these entities interact as a recurrence relation:

$$\begin{aligned} Simulation &= States_{0..n} \\ State_{t+1} &= f(State_t, Interactions_t) \\ Interactions_{t+1} &= \cup\, Behaviour_{agent}(state_t) \quad \forall\, agent \in Agents \end{aligned}$$

This formulation requires a function f, which merges the interactions of all the agents with the previous state to produce the next - we call this the *Reducer* function in our domain model, and it is a property of the simulation in general.

This analysis of the entities and relationships in the domain of agent-based modelling leads us to the following set of types from which our Haskell DSL can be composed:

```
data Simulation m s = {
  agents :: [Agent m s],
  initialState :: SimulationState s,
  reduce :: s -> [m] -> s
}

data Agent m s = {
  behaviour :: s -> [m]
}

runSim :: Simulation m s -> [s]
```

We define an algebraic data type representing a Simulation, parameterised over the type of the simulation state, and of the messages passed by agents, which denote their interactions. The simulation data type is composed of an initial state, a set of agents, and a reducer function. (We use Haskell's *record syntax* here to automatically define getter methods for each of the values). The Simulation type is parameterised over the types of both the state and the actions (the type variables m and s respectively). We also define the signature of a function that runs the simulation ($runSim$) - producing a list of its successive states.

This formulation accurately models the structure of an agent based model as Haskell types, however, there are a number of concerns that are not yet addressed. Given that agents are heterogenous and interacting, they must have some notion of identity which allows agents to identify each other (and the researcher running the simulation to identify the source of actions and their effects). In addition, (Macal &

North 2010) stipulates that agents should have the capacity for adaptive behaviour, which implies that they should have some sort of internal state; a capability that is not provided by our current definition. For this to be possible, each agent must have access to some internal state that allows it to store information about its environment and interactions, in order to adapt its behaviour in the future. Therefore, we extend our model with an additional type variable representing an agent's state, and a string representing the agent's identity[1]:

```

type AgentID = String

data Simulation a m s = {
  agents :: [Agent a m s],
  initialState :: s,
  reducer :: s -> [m] -> s
}

data Agent a m s = {
  agentID   :: AgentID
  behaviour :: (s, a) -> ([m], a)
}

runSim :: Simulation a m s -> [s]
```

We may also want to analyse the results of our simulation with reference to the messages passed between agents, instead of, or alongside the successive states of the simulation. For this reason, we define a new result type which encapsulates the states of the simulation and a log of the messages passed, and alter our $runSim$ function to return a value of this new type:

```
data SimResult m s = SimResult {
  messages :: [m],
  states :: [s]
}

runSim :: Simulation a m s -> SimResult ms
```

Finally, in order to achieve heterogeneity in our agents' interactions, we need to allow some degree of non-determinism in their behaviour. This necessarily means that it can no longer be represented by a pure function, and must instead be represented by an computation in the IO Monad, and the Simulation State as a value within the IO Monad (Wadler 1992). In fact, there are several aspects to our model which could be simplified by being modelled as Monadic actions - our agents' internal state can be represented by the State Monad, absolving us of the need to pass the state in as an extra argument to the behaviour function, and to return a tuple with the updated value, and removing the need for any 'plumbing' to pass the value back to the function on the next call. In addition, the logging of mes-

[1] We define a *type synonym* $AgentID$ for the String type here - this allows us to use a more meaningful type name in type signatures without the added overhead of boxing and unboxing the type, as would be the case with a $data$ or $newtype$ declaration. However, this is a slight trade-off, as it gives us no compile-time assurances that a given String is actually an AgentID or not.

sages passed between actors and the state can be achieved by maintaining a list of them in the Writer Monad, which allows them to be accumulated alongside the main state. We can capture the notion of a computation with all of these semantics using the *mtl* library, which draws on the work of Mark P Jones in (Jones 1995) to provide a *monad transformer* typeclass, which allows the semantics of monad-like structures to be composed into a single structure which is itself a monad.[2]. By applying the State and Writer transformers provided by *mtl* to the IO monad, we can construct a single type which will wrap our simulation state type, encapsulating all the above concerns; non-determinism, statefulness, and logging of messages:

```
newtype SimState a m s = SimState {
  unState :: StateT (MapAgentID a) (WriterT [m] IO) s
} deriving (Monad, MonadIO, MonadWriter [m], MonadState (Map AgentID
    a), Functor)
```

We wrap our monad transformer stack in a newtype[3] here, as this ensures we can hide the underlying implementation of the type by failing to export the constructor or the $unState$ accessor from its containing module. Thus, calling code only sees a Monad instance called SimState with certain features, and is agnostic towards its implementation. The deriving clause relies on an extension to the Haskell 98 standard provided by the Glasgow Haskell Compiler (Peyton Jones, Hall, Hammond, Cordy & Kevin 1992), enabling the compiler to automatically derive typeclass instances for a newtype whose member type is an instance of the same typeclass. Our new monad includes a Writer instance for the type $[m]$; a list of values of our message type, and a State value of type $Map\ AgentID\ a$, a map keyed by agent id, holding a value of the agent state type for each agent.

This design change requires some adjustment to the rest of our simulation API, giving us the set of types:

[2]Formally, a Monad is a type constructor `M a`, with the associated operations `return :: a -> M a` and `bind :: M a -> (a -> M a) -> M a`, where $return$ is both the left and right identity for $bind$, and that successive application of $bind$ is associative. In practical terms, this allows effectful computations to be sequenced with the $bind$ combinator (written >>= in Haskell) in a type safe manner. A Monad Transformer is a type constructor `T m a` with a single function `lift :: (Monad m) => m a -> T m a`, where the type `T m a` is also a monad by the above definition. This allows different types of effectful semantics to be ‘layered’ into a single monad.

[3]$newtype$ declarations allow the definition of a new algebraic data type with a single value, reducing the runtime overhead involved in boxing and unboxing algebraic datatypes by discarding the additional type information after the compile-time type-checking phase, treating the new type and its value as isomorphic. By contrast, conventional algebraic datatypes (defined with the ‘data’ keyword) with a single value are not isomorphic to their member type, due to a difference in their semantics where the member value is $\perp$. In short, newtypes are strict in their value, while data declarations are lazy. This distinction makes little practical difference to our implementation, meaning that the use of newtypes is more practical from an efficiency point of view.

```
type AgentID = String

newtype SimState a m s = SimState { unState :: StateT (Map AgentID a)
     (WriterT [m] IO) s } deriving (Monad, MonadIO, MonadWriter [m],
    MonadState (Map AgentID a), Functor)

data Agent a m s = Agent {
  agentID :: AgentID,
  behaviour :: s -> SimState a m (),
  initialAgentState :: a
}

data Simulation a m s = Simulation { agents :: [Agent a m s],
  initialState :: s,
  reduce :: s -> [m] -> SimState a m s
}

data SimResult m s = SimResult {states :: [s], log :: [m]}

runSim :: Simulation a m s -> IO (SimResult m s)
```

In addition to defining the parameters of the simulation itself, any user of our system will need to instrument the system in order to gather experimental data, and export that data in a useful format - this is an additional concern that will be incorporated into our simulation framework. Currently our simulation outputs a value of type $SimResult$ inside the IO monad, however, a user might reasonably want to instrument their simulation with information about the simulation's state at a particular point, or information about the messages passed between agents. To that end, we introduce a new type called $Logger$, which represents a instrumentation strategy for our simulation:

```
data Logger s m = Logger {
  setup :: s -> [m] -> IO ()
  run :: s -> [m] -> IO ()
}
```

Our logger type is parameterised over the types of the simulation's states and messages, and consists of a pair of functions - $setup$ - to be run at the start of the simulation only, and run, to be run on every step. Each function takes the current state and set of messages produced at that step, and performs an IO action - for example, echoing text to the screen or writing to a file. However, this definition still gives our user the responsibility of defining their own loggers - a requirement that seems unnecessary, as the method of instrumentation (writing states and messages to the screen or a file) is generic to all simulations, rather than a specific concern of any individual one. Instead, we provide generic logging functions which echo messages and states to STDOUT, and which output them to a file in the CSV format[4]. However, we still require the user to specify how each state or message should be serialized for output, and we provide a typeclass named $Loggable$ for this purpose.

[4]CSV is widely accepted as a format for transfer and storage of tabular data, supported by the vast majority of spreadsheet applications, as well as mathematical and statistical computing environments such as R and MATLAB

Loggable instances must implement a function called $output$ which takes an value of the loggable instance, and outputs a list of key and value pairs for properties to be logged, both of which are strings[5]:

```
type Field = String

class Loggable a where
  output :: a -> [(Field, Field)]

data (Loggable s, Loggable m) => Logger s m = Logger {
  setup :: s -> [m] -> IO (),
  run :: s -> [m] -> IO ()
}


runLoggers :: (Loggable s, Loggable m, Message m) => [Logger s m] ->
    SimResult m s -> IO (SimResult m s)
```

We also then restrict the parameters of our data type to instances of Loggable, ensuring that any attempt to log details of a non-loggable value results in a compile-time error. This allows us to define generic functions for logging states and messages to either STDOUT or a named file, as long as the user provides details of how values of the state and message types should be serialized, via their loggable instances. We also define the $runLoggers$ function, which takes a list of loggers, and applies them in turn to each successive state and set of messages in a value of type $SimResult$. This allows users of the simulation framework to take advantage of our generic logging functions, and to easily compose them with custom logger implementations designed for their own needs.

Our type definitions above give a simple, declarative syntax for defining and running agent-based models, and also abstracts away a large amount of generic functionality, allowing modellers using the system to concentrate on the implementation of the details of their model. While it does offer a DSL for the domain of agent-based modelling in general, this is of little use to researchers in other areas who simply want to use agent-based modelling as a tool to create simulations in their own respective domains of interest. In this sense, the facilities offered by our simulation DSL are still too generic to be considered an *ideal abstraction* (Hudak 1996) for users concerned with producing agent-based models. This, however, is one of the key advantages of embedding our simulation DSL within a host language - it affords the opportunity to build upon our framework, and compose it with other, complimentary abstractions in order to produce a language in which simulations of a particular domain can be defined in an expressive manner by domain experts. Our case-study provides an example of this process, yielding a high-level language for describing the interactions of simulated traders on a single security exchange. This process once again involves defining mappings between entities in our problem domain (financial markets) and our solution domain (agent-

[5]The $Field$ synonym for $String$ is provided by the Haskell CSV library which we make use of for serialization of data into CSV format.

based models). In this case, our problem domain is defined by the interactions between traders and the order book, which can be orders (messages from traders to the book) or trades (messages from the book to traders). These concepts have a natural and obvious mapping to the elements of our solution domain, The agents of our simulation correspond to traders, the state to the order book, and messages to orders and trades. Therefore, the state of our simulation can be entirely specified by end users in the language of financial markets, given that we provide an appropriate interface.

A financial market mediated by a limit-order book accommodates four different types of order, *bids* and *offers* (called *limit orders*), where the trader placing the order specifies a quantity of a security and a price point at which they are willing to buy or sell. The order is then held on the book until it is matched by an incoming *buy* or sell *sell* (*market orders*), which execute at the best price available on the book. Therefore, the type $Order$ in our simulation can be modelled by the union of market and limit orders, for both the buy and sell sides of the book. However, it would be advantageous for the specific type of order to be encoded in the type system, allowing the compiler to enforce aspects of their semantics that might otherwise cause a runtime error (for example, two orders on the same side of the book cannot be matched, and a market order cannot be matched with another market order). To this end, we define our order type as a parameterised type with two *phantom* type variables, which denote the type of order and side of the book respectively:

```haskell

data Buy
data Sell
class MarketSide a
instance MarketSide Buy
instance MarketSide Sell

data Market
data Limit
class OrderType a
instance OrderType Market
instance OrderType Limit

data (OrderType t, MarketSide m) => Order m t = Order TraderID Size (
    Maybe Price)
```

This definition encapsulates all the properties of an order, as well as encoding the order type and market side in the type signature. However, it has one important drawback - We use the Maybe monad[6] to denote the presence or absence of a price, but this property is uniquely determined by the type of order. For instance, a value `Order "TraderID" 150 Nothing` of type `Order Buy Limit` would be meaningless, as would a value `Order "TraderID" 150 (Just 50)` of type `Order Buy Market`. However, our current implementation does nothing to prevent this sort of inconsis-

[6]The type $Maybe$ is defined as `data Maybe a = Just a | Nothing` and denotes a value which may be present or absent. It is often described as a“type-safe null”.

tent state. We can improve this situation by explicitly providing data constructors for each order type:

```
data (OrderType t, MarketSide m) => Order m t = BuyOrder TraderID
    Size | SellOrder TraderID Size | BidOrder TraderID Size Price |
    OfferOrder TraderID Size Price
```

However, while we have improved matters by allowing the type of order to be denoted at the value level (through the choice of data constructor), we still have a potential inconsistency between the data and type levels - There is still nothing preventing a value `BuyOrder "TraderID" 150` having the type `Order Sell Limit`, for instance. Instead, we can use an extension to the Haskell standard provided by the Glasgow Haskell compiler to implement our order type as a *Generalised algebraic datatype* (Cheney & Hinze 2003), allowing us to define our order type by associating each data constructor with a type equation that specifies the type of the resulting value:

```
data Order side orderType where
  BuyOrder    :: TraderID -> Size ->          Order Buy Market
  SellOrder   :: TraderID -> Size ->          Order Sell Market
  BidOrder    :: TraderID -> Size -> Price -> Order Buy Limit
  OfferOrder  :: TraderID -> Size -> Price -> Order Sell Limit
```

This allows static checking of the semantics of our orders, including ensuring that each type of order has the appropriate value level members. A small amount of syntactic sugar also permits a more natural means of expressing orders, for example:

```
"trader1" ‘buys‘ quantity
"trader2" ‘bids‘ price ‘for‘ quantity
```

We also require a type representing a trade, which is simply a two-tuple of orders - one on the buy side, and one on the sell side. However, given that the types of these offers may differ, we define three data constructors, each denoting a particular type of trade that may take place:

```
data Trade =
  BuyTrade     (Order Buy Market) (Order Sell Limit)  |
  SellTrade    (Order Buy Limit)  (Order Sell Market) |
  CrossedTrade (Order Buy Limit)  (Order Sell Limit)
```

In addition, we define a generic trade constructor, which will dispatch on the Order constructor of its arguments and correct a trade with the correct constructor, as well as accessors for various properties of the trade which abstract away from the underlying types of the trade’s constituent orders:

```
makeTrade :: (OrderType a, OrderType b) => (Order Buy a) -> (Order
    Sell b) -> Trade

buySideTraderID :: Trade -> TraderID
sellSideTraderID :: Trade -> TraderID
tradedSize :: Trade -> Size
tradedPrice :: Trade -> Size
```

The type representing the limit order book must have the facility to maintain a list of open bid and offer orders, and to match these to incoming market orders. In addition, it should be possible to pre-seed the empty book with limit orders, in order that the price is stable at the start of the simulation. Finally, we require that the order book be parameterised by a function that applies a variable penalty to traders who place trades which have ill effects on liquidity or volatility in the book, attenuating their rate of trading. This function takes the state of the book before and after the trade, alongside the previous penalty applied to the trader in question, yielding a new penalty. The penalty itself is represented by the number of the next step of the simulation when the agent may trade. The book stores these penalties internally in a map, as well as keeping a record of all trades executed:

```

type PenaltyFunction = LOB -> LOB -> Penalty -> Penalty

data LOB = LOB {
  bids :: LimitOrderList Buy,
  offers :: LimitOrderList Sell,
  tradesDone :: [Trade],
  penalties :: Map TraderID Penalty,
  penaltyFunction :: PenaltyFunction
}

limitOrderBook :: LOB
withBids :: LOB -> [Order Buy Limit] -> LOB
withOffers :: LOB -> [Order Sell Limit] -> LOB
withPenaltyFunction :: LOB -> PenaltyFunction -> LOB
```

This allows the initial state of the order book to be defined with syntax such as the following:

```
book = limitOrderBook ‘withBids‘            ["trader1" ‘bids‘ 1500 ‘for
    ‘ 500]
                      ‘withOffers‘          ["trader2" ‘offers‘ 2500 ‘
                          for‘ 500]
                      ‘withPenaltyFunction‘ somePenaltyFunction
```

We also require the order book to expose accessors for various statistics to be used for instrumentation, and by trading agents to inform their trading strategy, including the best bid price, best offer price, depth and liquidity on either side of the book, and the price of the last executed trade. In addition, we require a function ”placeOrder” that takes an order and returns an updated book, along with either a list of executed Trades or a penalty value in the case where the trader placing the order is prohibited from trading. This requires a new type which wraps the possible response types, as well as a function that returns the traders to whom an order response is relevant:

```
data OrderResponse = TradeResponse Trade | PenaltyResponse TraderID
    Penalty
forTraders :: OrderResponse -> [TraderID]

placeOrder :: (MarketSide a, OrderType b) => Time -> LOB -> Order a b
     -> (LOB, [OrderResponse])
```

Note that the placeOrder function will accept orders of any type, and dispatches accordingly depending on its side of the book and order type. This ensures that the traders interface to the order book remains generic with regards to the type of order being placed, and well isolated from the underlying implementation.

The state type of our simulation consists of our order book, and also some type describing the underlying value of the security at a particular point in time. We model this underlying value as a function from time to price, and the state of the market itself as the union of an underlying value and an order book, as well as the time at that point, represented as an integer denoting the number of steps since the start of the simulation. Construction of the underlying value object is parameterised by a market sentiment and a starting price, where the market sentiment determines the function which calculates the price movement.

```
newtype Value = Value { getPrice :: Time -> Price }

data Sentiment = Calm | Choppy | Ramp | Toxic

underlyingValue :: Sentiment -> Price -> Value

data Market = Market {
  time :: Time,
  value :: Value,
  book :: LOB
}
```

Our traders form the agents in our simulation, and therefore must provide a unique string identifier, an initial state, and a behaviour which takes a market state value and yields a set of messages, optionally updating the initial state. The state value itself only needs to contain the agent's current inventory. The messages in our system may be orders or order responses, so we define a wrapper around these types encapsulating all the types of message within the system. We employ GHC's *existentially quantified types* (Laufer 1996) extension here to 'hide' the parameters to the order type, so that messages can be treated polymorphically by the simulation framework itself, whether they are orders (of any type), trades, or penalty messages:

```
  data MarketMessage = Response Time OrderResponse | forall a b . (
      MarketSide a, OrderType b) => Message Time (Order a b)

  newType TraderState = TraderState {
    inventoryLevel :: InventoryLevel
  }
```

This means that our simulation will have the type `Simulation TraderState MarketMessage Market`, and therefore our agents' behaviour will have the type `Market -> SimState MarketMessage ()` . As in (Clack 2011) and (Kirilenko et al. 2010), we define categories of trader behaviour according to their sensitivity to order imbalance and price volatility, their rate of increase in supply or demand of securities over the course of the simulation, their target inventory level, and the gap between their current inventory and target. We wish to provide an interface for specifying these trader types in a declarative manner, and then to instantiate a number of agents with the behaviour of the trader type to take part in the simulation. To this end, we define a $Trader$ data type, which encapsulates a trader's characteristic properties, and type synonyms for the various type of function by which our traders are parameterised. The inventory sensitivity function takes the current trader object, the trader's current inventory, the current underlying value and current middle price of the book. Volatility sensitivity functions take a Double representing the level of price volatility and returns a Double value which modulates the size and price of the order to be placed by the agent. Similarly, the Imbalance sensitivity takes an Int representing the difference in depth between the buy and sell sides of the book, and returns a Double value which is also used to calculate the price and size of the subsequent order.

```

type VolatilityFunction = Double -> Double
type ImbalanceFunction  = Int -> Double
type InventoryFunction = Trader -> Price -> Price -> Price

data :: Trader {
  traderID :: String,
  imbalanceFunction :: ImbalanceFunction,
  volatilityFunction :: VolatilityFunction,
  initialInventory :: Size,
  targetInventory :: Size,
  targetOrderSize :: Size,
  demandBias :: Int,
  orderSizeLimit :: Int,
  inventoryFunction :: InventoryFunction
}


makeTraders :: [(Int, Trader)] -> [Agent TraderState MarketMessage
    Market]
```

This type encapsulates all the logic that a trader needs to perform its function, however, without any additional syntax, it leads to a rather opaque and unwieldy syntax for defining trader types:

```
intermediary = Trader "intermediary" highImbalanceSensitivity
    lowVolatilitySesnsitivity 0 1200 200 0 3000 intermediaryInventory
```

This syntax is problematic, especially for users who are not necessarily experienced programmers, as the semantics of each argument are determined solely by their position in the statement, and the correct position for each argument can only be discovered by inspecting the code or accompanying documentation. As a result, it lacks expressiveness (in writing) and clarity (in reading) the trader definitions. A better approach relies on the *builder pattern* (Gamma, Helm, Johnson & Vlissides 1994), which is commonly exercised in DSL-based design as a means of creating a *fluent interface* (Ghosh 2010) for object creation. This allows new values to be created and configured in an incremental manner by successive function calls, enhancing readability and lucidity. For example:

```
intermediary = traderType "intermediary" highImbalanceSensitivity
    lowVolatilitySensitivity
                      ‘withInitialInventory‘  0
                      ‘withTargetInventory‘   1200
                      ‘withTargetOrderSize‘   200
                      ‘withDemandBias‘        0
                      ‘withOrderSizeLimit‘    3000
                      ‘withInventoryFunction‘ intermediaryInventory
```

This presents one difficulty, however - the value returned by the call `traderType "intermediary" highImbalanceSensitivity lowVolatilitySensitivity` cannot be a value of type Trader as defined above, as it lacks values for several fields. We could use the maybe monad to specify these fields as optional (as discussed above as part of our order model), however, as with orders, this offers us no static assurances that a given trader is fully specified with the correct attributes. Instead the use of phantom types (Iry 2010) once again offers a solution, allowing us to define a natural syntax for incremental object creation that also provides static guarantees that a given trader value is fully specified First, we define types wrapping each of the member values of the trader type which are incrementally added, using newtype declarations to ensure no runtime overhead in boxing and unboxing these types:

```
newtype HasInitialInventory = II Size
newtype HasTargetInventory = TI Size
newtype HasTargetOrderSize = TOS Size
newtype HasDemandBias = DB Int
newtype HasOrderSizeLimit = OSL Size
newtype HasInventoryFunction = IF InventoryFunction
```

We also define types indicating the absence of these properties:

```
data NoInitialInventory = NoII
data NoTargetInventory = NoTI
data NoTargetOrderSize = NoTOS
data NoDemandBias = NoDB
data NoOrderSizeLimit = NoOSL
data NoInventoryFunction = NoIF
```

We make each corresponding pair of these types an instance of some typeclass, which denotes the presence or absence of that particular value, for example:

```
class InitialInventory a
instance InitialInventory HasInitialInventory
instance InitialInventory NoInitialInventory
```

Using these types, we can then construct a datatype representing our traders which encodes the presence or absence of a given member value at the type level, as well as a type synonym denoting a fully-specified trader:

```haskell

data (InitialInventory ii, TargetInventory ti, TargetOrderSize tos,
    DemandBias db, OrderSizeLimit osl, InventoryFunc inf) =>
    TraderType ii ti tos db osl inf = Trader String ImbalanceFunction
     VolatilityFunction ii ti tos db osl inf

type Trader = TraderType HasInitialInventory HasTargetInventory
    HasTargetOrderSize HasDemandBias HasOrderSizeLimit
    HasInventoryFunction
```

We can then define our trader-building combinators as in the example above. Note also that this method also allows these combinators to be composed in any order, and ensures that the type signature of the yielded value reflects the presence or absence of its member values. For example:

```haskell

withInitialInventory :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf -> Size ->
    TraderType HasInitialInventory ti tos db osl inf
```

We also define getter functions for all of the attributes of a Trader, in order to ensure that the internal boxed representation of each value is invisible to code that makes use of the trader type.

4 Implementation

4.1 Simulation framework

As we have already seen, our design for our simulation framework specifies a set of types with which a particular simulation can be specified. All that remains is to implement the runSim function which takes a simulation and returns its results, obeying the semantics set out in (Clack 2011). This function has the type signature `runSim :: Simulation a m s -> IO (SimResult m s)`. In order to do this, we must create a list of all states in the simulation, before creating the result object. We first derive a function denoting a single step of the simulation, which calls the behaviour function for each agent in turn, before calling the reduce function to generate a new state. This function has the type signature `s -> SimState a m s`, which corresponds to an arrow in the Kleisli category (Moggi 1988) of our `SimState` monad. Thus, our complete sequence of simulation states can be shown as a successive binding of this function to our initial state:

```

step :: s -> SimState a m s

states :: [SimState a m s]
states = [initialState,
          initialState >>= step,
          initialState >>= step >>= step] --etc...
```

This is simply an iteration of the function `(>>= simStep)` on the initial state, and therefore can be expressed with the `iterate :: (a -> a) -> a -> [a]` function. We can then transform this to a list of state values contained within the SimState monad (`SimState a m [s]` as opposed to `[ SimState a m s]`) using the sequence combinator (`sequence :: (Monad m) => [m a] -> m [a]`), which sequentially executes each computation in a list of monadic computations, returning the result of each in a list which is itself wrapped in the monad. However, this approach is flawed in a fundamental way. By lazily constructing the monadic values at each step, and then subsequently sequencing them, each step of the simulation compute itself starting with the initial state value, rather than simply taking the result of the previous computation. This means that our simulation will run with $O(n^2)$ time complexity relative to the number of steps executed, rather than $O(n)$ time if the result of the previous computation was reused. In addition to this efficiency drawback, this also causes major problems for the semantics of our simulation. By re-executing each previous step of the simulation as input to subsequent steps, any effectful or non-deterministic actions will be re-executed, meaning that the resulting value may potentially differ from that yielded by a true iteration. To combat this, we define a new `iterateM` monad combinator with type `(Monad m) => (a -> m a) a -> m [a]` which performs a true iteration of the supplied monad arrow[7]:

[7]The initial implementation of the iterateM function given below was supplied by Brent Yorgey in a discussion on the haskell-beginners mailing list on August 15th, 2011.

```
iterateM :: (Monad m) => (a -> m a) -> a -> m [a]
iterateM f a = (a:) ‘liftM‘ (f a >>= iterateM f)
```

This works by creating a partially applied function that cons-es the passed state value onto some list, and lifting this into the monad, applying it to a value representing the tail of the list (created by calling the iterateM function recursively, bound to the result of applying the passed monadic action to the passed state). However, this approach, while preserving laziness, ensuring that impure computations are only computed once, and executing in $O(n)$ time has another flaw. Any attempt to execute the resulting monadic computation will not terminate, as the entire (infinite) list of successive applications of the arrow must be evaluated for their effects. As a result, we refine the iterateM function to take an additional argument specifying the number of iterations to compute. This may sacrifice some generality, but ensures that the resulting value can be used:

```
iterateM :: (Monad m) => Int -> (a -> m a) -> a -> m [a]
iterateM (-1) _ _ = return []
iterateM n    f a = (a:) ‘liftM‘ (f a >>= iterateM (n-1) f)
```

This implies that we must pass an integer representing the number of iterations to be executed into the runSim function, giving it the type: `Int -> Simulation a m s -> IO (SimResult m s)`. We also define our a function `simStep`, which takes a Simulation value, and returns a monadic function encapsulating a single step of the simulation, as described above. The returned function should execute each agent's behaviour function, and call the reducer function in order to produce an updated state. Since the type of our agent's behaviour is a monadic function that returns unit (`()`), we are only concerned about executing these actions for their effects. As a result, we can define another new monad combinator, `distM_` , which takes a list of monadic actions of type `(Monad m) => a -> m b` and composes them into a single action that passes its input into each action, executing them in sequence:

```
distM_ :: (Monad m) => [a -> m b] -> a -> m ()
distM_ fs x = sequence_ . map ($x) $ fs
```

We then use the `distM_` combinator in order to define our `simState` function:

```
simStep :: Simulation a m s -> a -> SimState a m s

simStep sim state = do
  let functions = map behaviour $ agents sim
  (_, messages) <- listen $ return >>= distM_ functions
  (reduce sim) state messages
```

This function uses the the `listen` function, which executes a computation in the writer monad, returning a tuple of its return value and all messages written during the course of its execution. We use this to collate all the messages yielded by our agents' behaviour functions for input into the reducer function, which then yields the new state. We can now use this function, along with our `iterateM` combinator to produce a list of the successive states of our simulation:

```
states :: Simulation a m s -> Int -> SimState a m [s]
states simulation iterations = iterateM iterations (simStep
    simulation) (initialState simulation)
```

All that remains, then, is to transform the list of simulation states into a value of type `IO (SimResult m s)`. In order to do this, we must unwrap that `StateT` and `WriterT` layers of our SimState monad, and lift a function that transforms the messages and states into a value of type SimResult. We use the `evalStateT` function to unwrap the `StateT` transformer, this takes a monad inside the `StateT` monad transformer, an initial state value, and returns the value of the underlying monad, discarding the final state value: `evalStateT :: (Monad m) => StateT s m a -> s -> m a`. We then unwrap the `WriterT` layer using the `runWriterT` function - rather than discarding the final writer value, this yields a tuple of the returned value and writer value within the underlying monad: `runWriterT :: WriterT w m a -> m (a, w)`. This then gives us a value of type `IO ([s], [m])`, which can be transformed into a value of type `IO (SimResult s m)` by lifting a function `(s, m) -> SimResult s m` into the IO Monad. We use the `uncurry` function (which takes a curried function with two arguments to an function with one tuple argument: `(a -> b -> c) -> (a,b) -> c`), applied to the `SimResult` constructor, to achieve this.

We also need to construct our initial state hash from the individual states of each agent. To do this we use the `fromList` function of the `Data.Map` module, which constructs a value of type `Map k v` from a list of type `[(k,v)]`. In our case, this list should have the agentID of each agent as the key, and their respective initial state as the value. We use the Arrow[8] (Hughes 2005) fanout combinator (`(&&&) :: (Arrow a) => a b c -> a b c′ -> a b (c, c′)`), and the Arrow instance for functions to achieve this.

This results in our final `runSim` implementation:

```
runSim :: Int -> Simulation a m s -> IO (SimResult m s)
runSim iterations sim = makeResult (simStep sim) (initialState sim)
  where makeResult step init  = liftM (uncurry SimResult) .
      runWriterT . flip evalStateT agentStates . unState $ states
          where agentStates           = M.fromList . map (agentID &&&
                initialAgentState) . agents $ sim
                 states               = iterateM iterations step init
```

4.2 Instrumentation and logging API

As described in the last section, we defined our instrumentation and logging API based around a `Logger` type and `Loggable typeclass`. We therefore need to define the `runLoggers` combinator that will apply the specified loggers to a simulation,

[8] Arrows provide an abstraction of computation similar to Monads, but where the type of the abstraction is parameterised by both the input and output type of the computation (Yorgey 2009). The `Control.Arrow` module provides an arrow instance for functions, and many useful combinators for composing functions, such as the fanout combinator used here.

and various default loggers for logging the simulation's states and messages to CSV files, and echoing them to the screen. Recall that our logger classes have the type:

```
data (Loggable s, Loggable m) => Logger s m = Logger {
  setup :: s -> [m] -> IO (),
  run :: s -> [m] -> IO ()
}
```

We therefore define an action in the IO monad that will take the output of our runSim function (`SimResult m s`) and a list of loggers, returning the result of the simulation and executing the loggers for their side-effects:

```
runLoggers :: (Loggable s, Loggable m) => [Logger s m] -> SimResult m
     s -> IO (SimResult m s)
```

each Logger's setup and run functions take a single state of the simulation, and a list of the messages generated in that state. This causes a slight setback, as we currently have no generic way of identifying the time at which a message was created. To combat this, we introduce another typeclass which messages in our simulation framework must implement, that provides a `messageTime` method returning an integer denoting the number of the iteration in which the message was generated, which we can then use to group the messages by iteration:

```
class Message m where
  messageTime :: m -> Int
```

Our `runLoggers` function itself uses the `distM_` combinator we derived for use in `runSim`, in order to distribute each state to each logger in the passed list. We use the `zip` function (`zip :: [a] -> [b] -> [(a,b)]`) to build a list of tuples of each simulation state and its corresponding messages, then call each loggers setup function with the first tuple, before iterating over the list of pairs using the `mapM_` combinator, calling each logger's run function on each pair in succession. This approach ensures that each logger executes in turn on each single iteration, rather than the first logger running on the whole list, followed by the second logger, etc. This has the result of ensuring, that should the simulation terminate early (due to user interaction or error), all the specified logs are preserved for inspection:

```
runLoggers :: (Loggable s, Loggable m, Message m) => [Logger s m] ->
    SimResult m s -> IO (SimResult m s)
runLoggers loggers result = do
  let pairs = zip (states result) (groupBy sameTime . messages $
      result)
  distM_ (map (uncurry . setup) loggers) $ head pairs
  mapM_ (distM_ $ map (uncurry . run) loggers) $ pairs
  return result
```

We specify four default loggers, for serializing and echoing to the screen both the simulation's states and messages. The echo instances simply print the values (ie the second element of each tuple returned) of a call to the `output` function on the state or message to STDOUT, while the csv loggers take a file path, and on setup ensure that no file already exists at that path, before writing a header line which prints the names of each loggable field (taken from the first element of each tuple returned by a call to output), before appending a line to the file for each state or message, on each call to the logger's run function. This provides a simple, yet flexible and generic means of logging the results of a simulation.

4.3 Securities trading DSL

4.3.1 The limit order book

The bulk of the implementation of our securities trading DSL is concerned with the effective modelling of our limit order book. As described in the previous section, this must hold lists of limit orders on either side of the book, for later execution when matched against an incoming market order. It must also apply rate-limiting penalties to traders specified by a user-defined function, and should have some ability to recover from a crossed book, by matching limit orders on both sides.

We define a new datatype called a `LimitOrderList` to hold the limit orders on either side of the book. This structure holds a list of orders at each price point, ensuring that orders are filled in order from best to worst price, and from earliest placed to last within each level. Each level of the list is represented by a value of type `OrderListLevel`, which has a price, and a *sequence* of orders.

```
data (MarketSide a) => OrderListLevel a = OrderListLevel {levelPrice
    :: Price, levelOrders :: Seq (Order a Limit)}
newtype (MarketSide a) => LimitOrderList a = LimitOrderList { levels
    :: [OrderListLevel a] }
```

Our use of a sequence (from the `Data.Sequence`) library here is crucial. Each order list level functions as a queue, with newly arrived orders being appended to the tail, and filled orders taken from the head, we need to use a doubly-linked data structure in order to ensure that locating the first item in the queue takes constant ($O(1)$), rather than linear ($O(n)$ time). The Sequence datatype provides a doubly linked list, and provides views of either side of the list, that may be used with GHC's *view patterns* (Wadler 1987) feature to allow pattern matches that match elements at either end of the list. For instance, the function `listSeqFromRight` produces a list of the elements of a given Sequence in reverse order, taking $O(n)$ time. However, thanks to Haskell's laziness, taking the last element (`head . listSeqFromRight`) takes only $O(1)$, as the entire list need not be evaluated in order to fetch the head:

```
listSeqFromRight :: Seq a -> [a]
listSeqFromRight (viewr -> EmptyR) = []
listSeqFromRight (viewr -> xs :> x) = x:listSeqFromRight(xs)
```

We also define instances of the orderability (`Ord`) and equality (`Eq`) type classes on our Order List Levels which encode some of the semantics of our order book. An order list level is equal to another when their price is the same, allowing us to use the `find` function in the `Data.List` package to find the appropriate level for any order inserted into the list, ensuring that each price point has a unique representation within the order list. The orderability of an order list level depends on the market side type by which it is parameterised - buy side levels are ordered from highest (best) price to lowest (worst), while sell side levels are ordered from lowest (best) price to highest (worst). This ensures that any difference in behaviour between order lists on the buy and sell side is fully specified in the lists type, and all the functions operating on the list can be written generically, deferring to the functions provided by the `Ord` instance where different behaviour is required. In particular, we can make use of the `insert` function in `Data.List` to ensure that the list of price levels is kept ordered from best to worst, regardless of the specific meaning of 'best' on either side of the market:

```
instance Eq (OrderListLevel a) where
  oll1 == oll2 = levelPrice oll1 == levelPrice oll2

instance Ord (OrderListLevel Buy) where
  compare oll1 oll2 = levelPrice oll2 ‘compare‘ levelPrice oll1

instance Ord (OrderListLevel Sell) where
  compare oll1 oll2 = levelPrice oll1 ‘compare‘ levelPrice oll2
```

This provides an underlying implementation of a list of limit orders that respects the ordering and equality semantics of orders on the book. We then define functions on this underlying data structure that allow orders to be inserted, deleted, and matched against market orders. These are then called by functions operating on the limit order book type itself, ensuring that the limit order list type itself is not visible to code which uses the limit order book, meaning that our limit order book's interface can remain consistent even if the underlying implementation changes.

We also guard against runtime errors by using functions from the `Safe` library in place of their unsafe alternatives, where a given operation could raise an exception. This is particularly important for functions that may take the head of an empty list, which will cause a runtime error. The `Safe` library uses the maybe monad to guard against runtime errors of this kind, allowing the programmer to specify an alternative path of execution, rather than relying on exception handling to recover from the error. For instance, the function `head` has type `[a] -> a`, while its safe equivalent, `headMay` has type `[a] -> Maybe a`. In combination with operations on the maybe monad, we can operate on the value returned by lifting arbitrary functions into the monad with the `liftM`, or its synonym, `fmap`[9], before 'unwrapping' the value and providing a default (returned in the case where the list was empty), using the `fromMaybe :: a -> Maybe a -> a` combinator. For example, the function `bestOrders` returns all the orders at the best price point in the book. A naive

[9]both `fmap` and and `listM` have the same semantics when operating on the Maybe monad, as it is an instance of both `Monad` and `Functor`. In fact, all monads are technically also functors, but not all monad instances in Haskell necessarily define a corresponding functor instance, so this equivalence can not be assumed in all cases.

implementation might be as follows:

```
bestOrders :: MarketSide a => LimitOrderList a -> [Order a Limit]
bestOrders = toList . levelOrders . head . levels
```

However, as we have discussed, this will thrown an exception if the list of levels in the order book is empty. A safer approach, which returns the empty list in the case where there are no price levels in the list is as follows:

```
bestOrders :: MarketSide a => LimitOrderList a -> [Order a Limit]
bestOrders = fromMaybe [] . fmap (toList . levelOrders) . headMay .
    levels
```

Here, the `headMay` function returns a an order level in the Maybe monad (`Maybe OrderListLevel a`). The function (toList . levelOrders) is then lifted into the monad, resulting in a type of `Maybe [Order a Limit]`, and finally the fromMaybe combinator returns either the value from the monad, in the case where it is `Just [Order a Limit]`, and returns the empty list if it is `Nothing`.

The limit order book type maintains a limit order list for either side of the book, and provides a unified interface for placing orders of all types through the function `placeOrder`. this dispatches based on the data constructor of the order passed to ascertain the type of order, and acts accordingly - placing the order in the appropriate limit order list where a limit order is passed, and executing the order against limit orders on the opposite side when a market order is passed. This function also guards against orders being placed by traders who have a penalty applied by taking the current iteration number as an argument and comparing it penalty values for each trader held in a map. Each penalty is represented by the number of the iteration in which the trader will be allowed to start trading again, and if the current time is less than this value, the order will be rejected. If the order succeeds, an updated limit order book, and a set of trades will be returned. This function also handles setting penalties for traders according the the function provided by the user when the simulation was defined.

The other concern which our order placing functionality must deal with is uncrossing the book, when the best offer order is lower priced than the best bid. In this case, we make a trade between the two orders at the mean price, and continue to do so until the book is no longer crossed. We define this procedure as a recursive function:

```
uncrossBook :: LOB -> (LOB, [Trade])
uncrossBook book = uncrossBook’ (book, [])
  where uncrossBook’ (b,t) | isCrossed b = uncrossBook’ (b’, trades’
      ++ t)
                           | otherwise   =  (b, t)
          where (bid, bids’)       = popBest (bids book)
                (offer, offers’)    = popBest (offers book)
                trades              = [makeTrade bid offer]
                trades’             = trades ++ tradesDone book
                b’                  = b { bids = bids’, offers =
                    offers’, tradesDone = trades’}
```

The actual pricing logic is not present here, as this is encapsulated within the `tradedPrice` function operating on the resulting trade. In the case where a trade consists of two limit orders, rather than a limit order and a market order, it returns the mean price of the two:

```
tradedPrice :: Trade -> Price
tradedPrice (BuyTrade     (BuyOrder _ _)    (OfferOrder _ p _))  = p
tradedPrice (SellTrade    (BidOrder _ p _)  (SellOrder  _ _))    = p
tradedPrice (CrossedTrade (BidOrder _ bp _)  (OfferOrder _ sp _)) = (
    bp + sp) ‘div‘ 2
```

4.3.2 Traders

Our method of defining individual traders strategies was set out in the previous section,and is largely based on the work in (Clack 2011). Each trader has a variety of numeric properties which define their propensity to supply or demand the security being traded, and their target inventory and order size. However, they also take various function arguments that define their sensitivity to properties of the market that we must define.

Traders can be defined as having low, medium or high sensitivity to both order imbalance and price volatility. Their sensitivity is modelled as a function that takes some metric representing the property in question, and returning a coefficient which our general trading algorithm can then use to determine order price, type and size. We define these as follows. For traders with a low sensitivity to a certain property, their coefficient for that property will be a constant 1. Those with medium sensitivity will have a coefficient given by the formula:

$$f(x) = 0.5 - \frac{1}{1 + e^{-x}}$$

Finally, those traders with high sensitivity have a coefficient given by the exponent of the value, with an arbirtary, configurable upper bound:

$$f(x) = \min(limit, e^{x})$$

These are expressed by the Haskell functions:

```
lowImbalanceSensitivity :: ImbalanceFunction
lowImbalanceSensitivity = const 1

mediumImbalanceSensitivity :: ImbalanceFunction
mediumImbalanceSensitivity = (0.5-) . sigmoid . fromIntegral

highImbalanceSensitivity :: ImbalanceFunction
highImbalanceSensitivity = min highSensitivityLimit . exp . fromIntegral

lowVolatilitySensitivity :: VolatilityFunction
lowVolatilitySensitivity = const 1

mediumVolatilitySensitivity :: VolatilityFunction
mediumVolatilitySensitivity = (0.5-) . sigmoid

highVolatilitySensitivity :: VolatilityFunction
highVolatilitySensitivity = min highSensitivityLimit . exp
```

Note we provide separate implementations for imbalance and volatility sensitivity, due to the need to cast the input to the imbalance functions from an integral value to a floating one.

Each trader type also has an inventory function specific to their trader type, which calculates the change to their target inventory based on the current underlying value of the security, the current mid price on the book and the trader's current inventory level. These functions are then used by a single, generic trading function to place orders on the market. This functionality is encapsulated within the makeAgent function, which turns a trader definition into a value of type `Agent TraderState MarketMessage Market`, with the appropriate behaviour defined by the trader definition.

```
makeTraders :: [(Int,Trader)] -> [Agent TraderState MarketMessage Market]
makeTraders pairs = pairs >>= uncurry fromPair
  where fromPair n trader = map (flip makeAgent trader) [1..n]

makeAgent :: Int -> Trader -> Agent TraderState MarketMessage Market
makeAgent n trader = Agent tid function initialState
  where tid = (traderID trader ++ "-" ++ show n)
        initialState = TraderState $ initialInventory trader
        function market = do
          (TraderState inventoryLevel) <- (flip (!) $ tid) ‘fmap‘ get
          let targetInv       = inventoryLevel + supplyDemand trader * time market
          let oSize           = orderSize trader market targetInv
          let oType           = orderType oSize (orderSizeLimit trader)
          let pricingStrategy = if oSize > (orderSizeLimit trader) then aggressive else neutral
          let oPrice          = pricingStrategy oType trader market
          placeOrder (time market) oType (abs oSize) oPrice tid
```

This function abstracts away all the functionality which is common to our different trader types, calling the functions set in our trader definition, where a specific behaviour is required. In addition, it delegates pricing of an order to a separate set of order pricing logic, which is also common to all traders. Pricing strategies are defined as either aggressive or neutral, depending on the willingness to trade exhibited by the trader, and the actual pricing function used also depends on the market outlook, defined as follows:

```
outlook :: Market -> MarketOutlook
outlook market | bb >= bo  && bo >= cv  = CrossedRising
               | bb >= cv  && cv >= bo = CrossedStable
               | cv >= bb  && bb >= bo  = CrossedFalling
               | bo >= bb  && bb >= cv  = Falling
               | bo >= cv  && cv >= bb = Stable
               | cv >= bo  && bo >= bb  = Rising
               where bb = bestBid $ book market
                     bo = bestOffer $ book market
                     cv = currentValue market
```

Having defined our traders' behaviour and provided a function to transform a trader type declaration into an agent capable of interacting with out simulation, we must define a reducer function for the market simulation, which will take all the orders produced by the traders, and produce an updated state of the market. This function must pass all the trades made by the traders in the order in which they were placed, and update the traders inventory, based on the trades executed as part of this process. It also writes the details of the trades made to the log for the purposes of monitoring and instrumentation. It finally creates a new state of the market with an updated order book, and increments the iteration counter by one:

```
updateMarket :: Market -> [MarketMessage] -> SimState TraderState
    MarketMessage Market
updateMarket last messages = do
  let tick               = time last
  let book'              = book last
  let (book'', responses) = placeOrders tick book' messages
  tell (map (Response tick) responses)
  agentState <- get
  put $ M.mapWithKey (updateStateForAgent responses) agentState
  return $ Market (tick + 1) (value last) book'' (bestBid book') (
      bestOffer book') (numOrders book')
```

5 Testing & Validation

In order to test our simulation framework and provide an example of its use, we employed it to produce a small simulation exploring the effects of rate attenuation on price volatility and liquidity in a limit order book. We trader definitions corresponding to the trader types outlined in (Clack 2011) and (Kirilenko et al. 2010), and simulated them trading in various market conditions, both with a book that applies no penalty to traders, and one with rate attenuation which increases exponentially with the amount of liquidity taken. Using the constructs defined in our trading DSL and simulation framework, we were able to adjust these parameters and record the results with ease. Our trader types were defined as follows, due to the time taken to run the simulation (our framework does not support concurrent execution), we used a limited number of traders that was lower than that specified in Kirilenko's paper:

	Sensitivity		Inventory		Order size			
Trader type	Imbalance	Volatility	Initial	Target	Target	Limit	Supply/demand bias	Number of traders
Intermediary	High	Low	0	1200	200	3000	0	3
High frequency trader	High	Low	0	800	100	3000	0	3
Fundamental buyer	Low	High	0	20000	200	300	200	2
Fundamental seller	Low	High	0	100	100	300	-200	2
Opportunistic trader	Medium	High	0	800	100	3000	0	4
Small trader	Medium	High	0	400	100	1	0	4

We ran the simulation with varying market conditions - with an underlying value ramping down, and with a choppy underlying value, specified as a sine wave. The attenuation functions we tested were defined as follows:

```

exponentialBackoffPenalty lobBefore lobAfter penaltyBefore =
    penaltyBefore + (floor . exp . fromIntegral $ liquidityTaken)
        where liquidityTaken = buySideLiquidity lobBefore +
            sellSideLiquidity lobBefore - buySideLiquidity lobAfter -
            sellSideLiquidity lobAfter

exponentialBackoffPenaltyWithLimit limit lobBefore lobAfter
    penaltyBefore = penaltyBefore + min limit (floor . exp .
    fromIntegral $ liquidityTaken)
        where liquidityTaken = buySideLiquidity lobBefore +
            sellSideLiquidity lobBefore - buySideLiquidity lobAfter -
            sellSideLiquidity lobAfter
```

The first applies a penalty which is exponential in the amount of liquidity taken from the book, the second is the same, but with the addition of an upper bound on this penalty. This was introduced as, under some conditions, the first limiting function effectively banned some agents from trading outright. We measured the outcome of this experiment taking the standard deviation of traded prices as an indicator of price volatility, alongside the maximum drawdown in liquidity over the course of the simulation:

Market sentiment	Penalty function	Number of steps	Volatility	Liquidity drawdown
Choppy	None	500	1274	-35
Ramp	None	500	428	-27
Choppy	Exponential	500	1128	-110
Ramp	Exponential	500	342	-69

These results show a small improvement in both volatility and liquidity (a negative mean drawdown denotes an average increase in liquidity on each successive trade), in line with our expectations, but the significance of these levels of improvement is minimal. More research is needed to tune the penalty function to provide an optimal decrease in volatility and increase in liquidity, however, these results validate the fact that our framework is capable of simulating the actions of traders in a limit order book.

6 Further work

Our framework, although offering a promising environment for performing agent-based simulation could be enhanced in several ways. In particular, it suffers from the drawback that it currently only runs on a single core of a single machine, meaning that running a single complex simulation can take some time, as the time taken increases linearly with the number of steps simulated, and the number of agents used. However, it would be possible to adapt our framework to take advantage of parallelism - the pure functional style of programming which Haskell encourages (and which we have used) is particularly well suited to the task of writing concurrent programs, as the lack of shared state reduces the need for explicit control of access to shared resources. The actor model of concurrent processing (as implemented in the Erlang language) has been cited as a suitable environment for agent-based simulation (Agha 1985), however this suffers one drawback in that the order of execution of each agent's behaviour is non-deterministic, meaning that simulations that rely on a well-defined order of operations will suffer.

There are, however, a number of existing abstractions for concurrent and distributed programming which would not present this drawback including Hoare's Communicating Sequential Processes (Hoare 1978). CSP defines a calculus of intercommunicating processes which can pass messages between each other, and operators by which processes can be sequenced and parallelised, all with a well defined event ordering semantics, which is essential for simulations that rely on a deterministic order of operations. In this case. In our case, we could define a process which executes all the agents behaviours in parallel, and sequence this with the simulation's reduce function, achieving parallelism in execution of the agents' behaviours, without sacrificing a well-defined ordering of events in the system. The CSP model has been implemented as a Haskell library named CHP (Brown 2008), which could be integrated easily into our existing simulation framework.

In addition, a number of enhancements could be made to enhance the usefulness of our system for end users, the ability to produce more complex output from a simulation (for instances, summary statistics or visualisations) using the same declarative syntax as we have proposed for defining simulations would reduce the need for other specialised tools.

7 Summary & Conclusion

This project has introduced a framework for describing agent-based simulations and a DSL for describing interactions in a limit-order securities market, which have been used successfully to investigate the effects of rate-attenuation on levels of volatility and liquidity in a simple market. Despite the fact that our case-study simulation's results were inconclusive, it validates the design choices made at the outset of the project, in offering an expressive, easy to use environment for defining and running such simulations.

Appendix A: Users manual

Our framework will run on any POSIX-compatible UNIX machine with the Haskell Platform installed - binaries for various platforms and installation instructions for this are available at `http://hackage.haskell.org/platform/`. It also requires the 'cabal-dev' build and depenency management tool, this can be installed using the cabal package manager which is included in the Haskell Platform - simply run the command `cabal install cabal-dev` from a shell prompt.

Before first running our simulation code, use cabal-dev to install the project dependencies by running the command `cabal-dev install-deps` from within the project directory. Finally, to run the simulation, execute the r#run.sh shell script within the project directory in order to compile and run the simulation.

To use the Simulation DSL in your own work, simply import the modules needed from the project. These are:

`Simulation.Simulation` – Exports the `Simulation`, `Agent`, `SimState` and `SimResult` datatypes, the `Message` typeclass, and the `runSim` function which executes a simulation, returning its results. Define values of these types in order to construct and run a simulation

`Simulation.Loggers` – defines the `Logger` type and `Loggable` typeclass, as well as the default logging functions `echoMessages`, `echoStates`, `logStates`, and `logMessages`.

Appendix B: Glossary of common Haskell concepts

General syntax

Haskell has a number of syntactic conventions that seem odd to those who are unfamiliar with the language, I attempt to outline a few of these here for reference. A more complete list is available at `http://www.haskell.org/haskellwiki/Reference_card` if required. Comments are prefixed by a pair of hyphens (`--`).

7.0.3 Function definition

Functions in Haskell are defined in the same way as values, with the equals (`=`) symbol. In fact, there is really no difference between the two - a constant value is treated the exact same way as a function which does not take any arguments. Named arguments to a function are placed before the equals sign:

```
four = 4 -- A constant value
plusTwo x = x + 2 -- A function with a single named argument
```

Haskell is a strongly typed language, but has an advanced type-inference system which allows functions to be written without a type signature. However, type-signatures are included liberally throughout the project, both to aid readability and understanding, and occasionaly in places where the compiler is not able to inference the type of a function. The syntax for type signatures is as follows:

```
four :: Int -- denotes a value of type Int
plusTwo :: Int -> Int -- denotes a function that takes an Int
    argument and returns another Int
add :: Int -> Int -> Int -- denotes a function that takes two Int
    arguments and returns an Int
length :: [a] -> Int -- denotes a function that takes a list of any
    type, and returns an Int.
```

In the above examples, the symbol (`::`) denotes the 'has type of' relation between a value or function name and a type. The arrow (`->`) represents a mapping (a function type), and the square brackets `[ ]` denote a list of a certain type. Note that we can employ type variables (with an initial lowercase letter) to signal that any type is acceptable in a certain position in the type signature.

7.0.4 Function application

Functions are applied to their arguments by juxtaposition, and function application is left-associative. So, `plusTwo 5` would yield the value `7`, for instance. Functions can also be composed together using the (`.`) operator, which has lower precedence than function application. expressions in parentheses have higher precedence than those outside, as is common in many languages, however, idiomatic Haskell style favours the use of the (`$`) operator over parentheses - this is a synonym for function application with lower precedence than any other operator, and can be used in place of parentheses in many situations. For example: `plusTwo $ length "A string"` is equivalent to `plusTwo (length "A string")`.

7.0.5 Type declarations

New datatypes can be defined with the data keyword - which takes a type constructor (the name of the type), a data constructor and set of member types: `data Coordinate = Coord Int Int`. Data types can have multiple data constructors, for example: `data Coordinate = Cartesian Int Int | Polar Double Int`. In addition, it's possible to parameterise a data type over some other type, for instance the datatype `List a = Cons a | Nil` denotes a cons list of values of any type `a`. Finally, Haskell also includes a features known as 'record syntax', which allows member values of a type to be given names, automatically providing getter functions for the values. For instance, `data Coordinate { x :: Int, y :: Int}` defines a new Coordinate type with two integer members, as well as the accessor functions `x :: Coordinate -> Int` and `y :: Coordinate -> Int`. More information on type declaration syntax is available at `http://en.wikibooks.org/wiki/Haskell/Type_declarations`.

Type classes

Haskell's type classes fulfil a function similar to interfaces in an object-oriented language, defining a set of functions that can apply to multiple types, and a set of constraints that implementing types must fulfil. Classes are defined with the `class` keyword as follows:

```
class Comparable a where
  -- these functions must be defined in an instance declaration:
  lessThanOrEqual :: a -> a -> Bool
  equal           :: a -> a -> Bool

  -- these functions are available for 'free' on any instance:
  greaterThan       = not lessThanOrEqual
  lessThan          = lessThanOrEqual && not equal
  greaterThanOrEqual = equal && not lessThanOrEqual
```

Instances of typeclasses are defined as follows:

```
instance Comparable Int where
  lessThanOrEqual i1 i2 = i1 <= i2
  equal           i1 i1 = i1 == i2
```

Now, functions and types can be defined that constrain their type variables to members of a certain class. These constraints are expressed in the type signature using the `(=>)` arrow. For instance: `maximum ::`

`(Comparable a) => [a] -> a`. More information on typeclasses is available at `http://www.haskell.org/tutorial/classes.html`.

Monads

Monads are an abstraction that permeate a lot of non-trivial Haskell code, as they are used extensively to structure effectful computations, such as those involving mutable state, non-determinism and IO. Formally, a Monad is any type which is an instance of the following typeclass:

```
   class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b       --pronounced 'bind'
\end {code}

Instances of the Monad typeclass must satisfy the following laws:

\begin{code}
  return a >>= f = f a                    -- return is the left-identity of (>>=)
  m >>= return   = m                      -- return is the right-identity of (>>=)
  m >>= f >>= g  = m >>= (\x -> f x >>= g) -- composition of monadic functions is associative
```

This may seem rather abstract, but it has some useful consequences, particularly in that the bind (`>>=`) operation allows sequencing of effectful computations. Haskell also provides various other Monad combinators (the most significant being `>=>` and `>>`), and a special syntax for sequencing monadic operations that's closer to the style of a traditional imperative language, known as 'do notation':

```
  (>>)  :: m a -> m b -> m b                      -- sequences two monadic actions (as >>=), but discards the returned value of the first, executing it only for its effects
  (>=>) :: (a -> m b) -> (b -> m c) -> a -> m c   -- composes two monadic actions together.

  --"do notation":

  displayCharCode = do
    putStrLn "Press a key:"
    char <- getChar
    putStrLn "You pressed the key with code:" ++ show (ord char)

  -- This is equivalent to displayCharCode = putStrLn "Press a key:" >> getChar >>= (\char -> putStrLn "You pressed the key with code: " ++ show (ord char))
```

There is more information on the role of Monads in Haskell available at `http://www.haskell.org/haskellwiki/Monad`, and (Yorgey 2009), which also provides information on various other useful Haskell constructs. (O'Sullivan, Goerzen & Stewart 2008) also provides a useful, practical introduction to the Haskell language as a whole.

Appendix C: Code listing

Listing 1: Main.hs

```
import Simulation.Simulation
import Simulation.Market hiding (book, value)
import Simulation.Order
import Simulation.Loggers
import Simulation.Market.LoggableInstances
import Simulation.Traders
import Simulation.Traders.InventoryFunctions
import Simulation.Traders.SensitivityFunctions
import Simulation.Traders.Types
import Simulation.LimitOrderBook
import Control.Monad
import Debug.Trace

ticks = 150

value = underlyingValue Ramp 2000 ticks

resultsDirectory= "../../results/"

testName = "ramp-noPenalty-lowNTraders-150steps-3"

exponentialBackoffPenaltyWithLimit limit lobBefore lobAfter penaltyBefore = penaltyBefore + min limit (floor . exp . fromIntegral $ liquidityTaken)
        where liquidityTaken = buySideLiquidity lobBefore + sellSideLiquidity lobBefore - buySideLiquidity lobAfter - sellSideLiquidity lobAfter

exponentialBackoffPenalty lobBefore lobAfter penaltyBefore = penaltyBefore + (floor . exp . fromIntegral $ liquidityTaken)
        where liquidityTaken = buySideLiquidity lobBefore + sellSideLiquidity lobBefore - buySideLiquidity lobAfter - sellSideLiquidity lobAfter

book = limitOrderBook ‘withBids‘            ["fundamentalBuyer-1" ‘bids‘ 1500 ‘for‘ 500]
                      ‘withOffers‘          ["fundamentalSeller-1" ‘offers‘ 2500 ‘for‘ 500]
                      ‘withPenaltyFunction‘ noPenalty

intermediary = traderType "intermediary" highImbalanceSensitivity lowVolatilitySensitivity
                        ‘withInitialInventory‘  0
                        ‘withTargetInventory‘   1200
                        ‘withTargetOrderSize‘   200
                        ‘withDemandBias‘        0
                        ‘withOrderSizeLimit‘    3000
                        ‘withInventoryFunction‘ intermediaryInventory

hfTrader = traderType "hfTrader" highImbalanceSensitivity lowVolatilitySensitivity
                        ‘withInitialInventory‘  0
                        ‘withTargetInventory‘   800
                        ‘withTargetOrderSize‘   100
                        ‘withDemandBias‘        0
                        ‘withOrderSizeLimit‘    3000
                        ‘withInventoryFunction‘ hfInventory
```

```
fundamentalBuyer = traderType "fundamentalBuyer" lowImbalanceSensitivity highVolatilitySensitivity
                        `withInitialInventory`  0
                        `withTargetInventory`   20000
                        `withTargetOrderSize`   200
                        `withDemandBias`        200
                        `withOrderSizeLimit`    300
                        `withInventoryFunction` fundamentalBuyerInventory

fundamentalSeller = traderType "fundamentalSeller" lowImbalanceSensitivity highVolatilitySensitivity
                        `withInitialInventory`  0
                        `withTargetInventory`   100
                        `withTargetOrderSize`   100
                        `withDemandBias`        (-200)
                        `withOrderSizeLimit`    300
                        `withInventoryFunction` fundamentalSellerInventory


opportunisticTrader = traderType "opportunisticTrader" mediumImbalanceSensitivity highVolatilitySensitivity
                        `withInitialInventory`  0
                        `withTargetInventory`   800
                        `withTargetOrderSize`   100
                        `withDemandBias`        0
                        `withOrderSizeLimit`    3000
                        `withInventoryFunction` opportunisticTraderInventory

smallTrader = traderType "smallTrader" mediumImbalanceSensitivity highVolatilitySensitivity
                        `withInitialInventory`  0
                        `withTargetInventory`   400
                        `withTargetOrderSize`   100
                        `withDemandBias`        0
                        `withOrderSizeLimit`    1
                        `withInventoryFunction` smallTraderInventory

traders = [(3, intermediary),         -- 3 / 11
           (3, hfTrader),             -- 3 / 1
           (2, fundamentalBuyer),     -- 2 / 79
           (2, fundamentalSeller),    -- 2 / 80
           (4, opportunisticTrader),  -- 4 / 363
           (4, smallTrader)]          -- 4 / 430

loggers = [echoStates, echoMessages, logStates (resultsDirectory ++ "states-" ++ testName ++ ".csv"), logMessages (resultsDirectory ++ "messages-" ++ testName ++ ".csv")]

main = runLoggers loggers =<< (runSim ticks $ Simulation (makeTraders traders) (makeMarket value book) updateMarket)
```

Listing 2: Simulation/Constants.hs

```
module Simulation.Constants where
  topOfBookThreshold = 0.05
  highSensitivityLimit = 40.0
```

Listing 3: Simulation/LimitOrderBook.hs

```haskell
{-# LANGUAGE GADTs #-}
module Simulation.LimitOrderBook(LOB(), limitOrderBook, noPenalty,
    withBids, withOffers, withPenaltyFunction, bestBid, numOrders,
    bestOffer, buySideLiquidity, sellSideLiquidity, buySideDepthNearTop
    , sellSideDepthNearTop, buySideDepth, sellSideDepth,buySideLevels,
    sellSideLevels, midPrice, lastTradedPrice, placeOrder, isCrossed)
    where
  import Simulation.Types
  import Simulation.Order hiding (bids, offers)
  import Simulation.OrderResponse
  import Simulation.Trade
  --import Simulation.Traders
  import Simulation.LimitOrderList hiding (empty, insert, numOrders)
  import qualified Simulation.LimitOrderList as LOL (empty, insert,
      numOrders)
  import Data.Map (Map)
  import qualified Data.Map as M
  import Control.Applicative hiding (empty)
  import Safe
  import Data.Maybe (fromMaybe)
  import Debug.Trace

  type PenaltyFunction = LOB -> LOB -> Penalty -> Penalty

  data LOB = LOB {
    bids :: LimitOrderList Buy,
    offers :: LimitOrderList Sell,
    tradesDone :: [Trade],
    penalties :: Map TraderID Penalty,
    penaltyFunction :: PenaltyFunction
  }

  noPenalty = (flip $ const . flip const)

  limitOrderBook = LOB LOL.empty LOL.empty [] M.empty noPenalty

  withBids :: LOB -> [Order Buy Limit] -> LOB
  withBids book orders = foldl place book orders
    where place book order = book { bids   = LOL.insert (bids book)
        order }

  withOffers :: LOB -> [Order Sell Limit] -> LOB
  withOffers book orders = foldl place book orders
    where place book order = book { offers = LOL.insert (offers book)
        order}

  withPenaltyFunction :: LOB -> PenaltyFunction -> LOB
  withPenaltyFunction book function = book {penaltyFunction = function}

  bestBid :: LOB -> Price
  bestBid = bestPrice . bids

  bestOffer :: LOB -> Price
  bestOffer = bestPrice . offers

  numOrders :: LOB -> Int
  numOrders book = numBids book + numOffers book

  numBids :: LOB -> Int
  numBids = LOL.numOrders . bids
```

```haskell

numOffers :: LOB -> Int
numOffers = LOL.numOrders . offers

buySideLiquidity :: LOB -> Int
buySideLiquidity = liquidity . bids

sellSideLiquidity :: LOB -> Int
sellSideLiquidity = liquidity . offers

buySideDepth :: LOB -> Int
buySideDepth = depth . bids

sellSideDepth :: LOB -> Int
sellSideDepth = depth . offers

buySideDepthNearTop :: LOB -> Double -> Int
buySideDepthNearTop = depthNearTop . bids

sellSideDepthNearTop :: LOB -> Double -> Int
sellSideDepthNearTop = depthNearTop . offers

buySideLevels :: LOB -> Int
buySideLevels = numLevels . bids

sellSideLevels :: LOB -> Int
sellSideLevels = numLevels . offers

midPrice :: LOB -> Price
midPrice book = floor $ fromIntegral (bestBid book + bestOffer book) / 2

lastTradedPrice :: LOB -> Price
lastTradedPrice = fromMaybe 0 . fmap tradedPrice . headMay . tradesDone

placeOrder :: (MarketSide a, OrderType b) => Time -> LOB -> Order a b -> (LOB, [OrderResponse])
placeOrder tick book order | tick <= penaltyForTrader  = (book,  [PenaltyResponse trader penaltyForTrader])
                           | otherwise                 = (book'', map TradeResponse trades'')
  where trader            = traderID order
        penaltyForTrader  = M.findWithDefault 0 trader (penalties book)
        trades''          = trades ++ trades'
        (book'', trades') = uncrossBook $ book' {penalties = penalties'}
        penalties'        = M.insert trader newPenalty (penalties book)
        newPenalty        = penaltyFunction book book book' penaltyForTrader
        (book', trades)   = bookAndTrades
        bookAndTrades :: (LOB, [Trade])
        bookAndTrades     = case order of
          (BidOrder   _ _ _) ->  (book { bids   = LOL.insert  (bids book) order}, [])
          (OfferOrder _ _ _) ->  (book { offers = LOL.insert  (offers book) order}, [])
          (BuyOrder   _ _) ->  executeBuy  order book
          (SellOrder  _ _) ->  executeSell order book

```

```haskell
executeBuy :: (Order Buy Market) -> LOB -> (LOB, [Trade])
executeBuy order book = (book’, trades)
  where book’                  = book {offers = offers’, tradesDone = trades}
        (offers’, offerOrders) = toFill (offers book) order
        trades                 = map (makeTrade order) offerOrders ++ tradesDone book


executeSell :: (Order Sell Market) -> LOB -> (LOB, [Trade])
executeSell order book = (book’, trades)
  where book’              = book {bids = bids’, tradesDone = trades}
        (bids’, bidOrders) = toFill (bids book) order
        trades             = map (flip makeTrade order) bidOrders ++ tradesDone book

isCrossed :: LOB -> Bool
isCrossed book = (not . elem 0 $ [numBids, numOffers] <*> return book) && bestBid book > bestOffer book

uncrossBook :: LOB -> (LOB, [Trade])
uncrossBook book = uncrossBook’ (book, [])
  where uncrossBook’ (b,t) | isCrossed b = uncrossBook’ (b’, trades’ ++ t)
                           | otherwise   =  (b, t)
          where (bid, bids’)       = popBest (bids book)
                (offer, offers’)    = popBest (offers book)
                trades              = [makeTrade bid offer]
                trades’             = trades ++ tradesDone book
                b’                  = b { bids = bids’, offers = offers’, tradesDone = trades’}
```

Listing 4: Simulation/LimitOrderList.hs

```haskell
{-# LANGUAGE FlexibleInstances, FlexibleContexts, UndecidableInstances, ViewPatterns #-}
module Simulation.LimitOrderList (LimitOrderList(), empty, isEmpty, bestPrice, numOrders, bestOrders, worstPrice, worstOrders, insert, delete, liquidity, depth, depthNearTop,  numLevels, toFill, popBest, orders) where
  import Prelude hiding (last, length, filter, zip, scanl, drop, splitAt, concat, sum, null, foldl)
  import qualified Data.List as L
  import Data.List ((\\))
  import Data.Maybe (fromMaybe, listToMaybe)
  import Simulation.Types
  import Simulation.Order
  import Data.Sequence hiding (null, empty, length)
  import qualified Data.Sequence as S (null, empty, length)
  import Data.Foldable
  import Safe
  import Debug.Trace

  data OrderListLevel a = OrderListLevel {levelPrice :: Price, levelOrders :: Seq (Order a Limit)}

  instance Show (OrderListLevel a) where
    show (OrderListLevel p _) = "OrderListLevel: " ++ show p

  instance Eq (OrderListLevel a) where
    oll1 == oll2 = levelPrice oll1 == levelPrice oll2
```

```haskell

instance Ord (OrderListLevel Buy) where
  compare oll1 oll2 = levelPrice oll2 `compare` levelPrice oll1

instance Ord (OrderListLevel Sell) where
  compare oll1 oll2 = levelPrice oll1 `compare` levelPrice oll2

newtype LimitOrderList a = LimitOrderList { levels :: [OrderListLevel a] }

instance (Show (Order a Limit)) => Show (LimitOrderList a) where
  show = ("LimitOrderList" ++) . show . map (show . levelOrders) . levels

empty :: MarketSide a => LimitOrderList a
empty = LimitOrderList []

isEmpty :: MarketSide a => LimitOrderList a -> Bool
isEmpty = L.null . levels

numOrders :: MarketSide a => LimitOrderList a -> Int
numOrders = L.length . orders

orders :: MarketSide a => LimitOrderList a -> [Order a Limit]
orders = concat . map ordersForLevel . levels

bestPrice :: MarketSide a => LimitOrderList a -> Price
bestPrice = fromMaybe 0 . fmap levelPrice . headMay . levels

bestOrders :: MarketSide a => LimitOrderList a -> [Order a Limit]
bestOrders = fromMaybe [] . fmap (toList . levelOrders) . headMay . levels

popBest :: (Eq (Order a Limit), Ord (OrderListLevel a), MarketSide a) => LimitOrderList a -> (Order a Limit, LimitOrderList a)
popBest orderlist       = (order, orderlist')
  where order           = head . bestOrders $ orderlist
        orderlist'      = delete orderlist order

worstPrice :: MarketSide a => LimitOrderList a -> Price
worstPrice = levelPrice . L.last . levels

worstOrders :: MarketSide a => LimitOrderList a -> [Order a Limit]
worstOrders = toList . levelOrders . L.last . levels

insert :: (Ord (OrderListLevel a), MarketSide a) => LimitOrderList a -> Order a Limit -> LimitOrderList a
insert orderList order = LimitOrderList $ L.insert level' (L.delete level $ levels orderList)
  where level = levelFor orderList order
        level' = OrderListLevel (levelPrice level) (order <| levelOrders level)

delete :: (Eq (Order a Limit), Ord (OrderListLevel a), MarketSide a) => LimitOrderList a -> Order a Limit -> LimitOrderList a
delete orderList order = LimitOrderList levels''
  where level   = levelFor orderList order
        level'  = OrderListLevel (levelPrice level) (seqDelete order $ levelOrders level)
        levels' = L.delete level (levels orderList)
        levels'' | S.null $ levelOrders level' = levels'
```

```haskell
                  | otherwise                   = L.insert level' levels'

liquidity :: MarketSide a => LimitOrderList a -> Size
liquidity = sum . map size . orders

depth :: MarketSide a => LimitOrderList a -> Size
depth = sum . map size . bestOrders

depthNearTop :: MarketSide a => LimitOrderList a -> Double -> Size
depthNearTop list percent = sum . map size . ordersInTop list $ percent

numLevels :: MarketSide a => LimitOrderList a -> Int
numLevels = L.length . levels

toFill :: (Eq (Order a Limit), Ord (OrderListLevel a), MarketSide a, MarketSide b, OrderType c) => LimitOrderList a -> Order b c -> (LimitOrderList a, [Order a Limit])
toFill list order = (list'', matchingOrders)
  where list'           = foldl delete list matchingOrders
        list''          = fromMaybe list' $ insert list' `fmap` partialOrder'
        partialOrder    = orderPartiallyFilledByOrder list order
        partialOrder'   = updateSize partialFillSize `fmap` partialOrder
        partialFillSize = size order - (sum . map size . ordersTotallyFilledByOrder list $ order)
        matchingOrders  = ordersMatchingOrder list order

levelFor :: (Ord (OrderListLevel a), MarketSide a) => LimitOrderList a -> Order a Limit -> OrderListLevel a
levelFor list order = fromMaybe (OrderListLevel (price order) S.empty) $ L.find ((== price order) . levelPrice) . levels $ list

ordersForLevel :: MarketSide a => OrderListLevel a -> [Order a Limit]
ordersForLevel = listSeqFromRight . levelOrders
  where listSeqFromRight (viewr -> EmptyR) = []
        listSeqFromRight (viewr -> xs :> x) = x:listSeqFromRight(xs)

ordersInTop :: MarketSide a => LimitOrderList a -> Double -> [Order a Limit]
ordersInTop list percent = concat . map (toList . levelOrders) . L.filter (inTop percent) . levels $ list
  where inTop :: Double -> (OrderListLevel a -> Bool)
        inTop p | spread > 0 = (>= (fromIntegral . floor $ bottomPrice + spread*p)) . levelPrice
                | otherwise  = (<= (fromIntegral . floor $ bottomPrice + (1-p)*abs spread)) . levelPrice
          where spread = fromIntegral $ bestPrice list - worstPrice list
                bottomPrice = fromIntegral . worstPrice $ list

ordersMatchingOrder :: (MarketSide a, MarketSide b, OrderType c) => LimitOrderList a -> Order b c -> [Order a Limit]
ordersMatchingOrder list order = takeWhileAccumulating (>=0-s) (\s o -> s - size o) s os
  where s  = size order
        os = orders list

ordersTotallyFilledByOrder :: (MarketSide a, MarketSide b, OrderType c) => LimitOrderList a -> Order b c -> [Order a Limit]
```

```
ordersTotallyFilledByOrder list order = takeWhileAccumulating (>=0)
    (\s o -> s - size o) s os
  where s = size order
        os = orders list

orderPartiallyFilledByOrder :: (Eq (Order a Limit), MarketSide a,
    MarketSide b, OrderType c) => LimitOrderList a -> Order b c ->
    Maybe (Order a Limit)
orderPartiallyFilledByOrder list order = listToMaybe $
    ordersMatchingOrder list order \\ ordersTotallyFilledByOrder list
     order


takeWhileAccumulating :: (b -> Bool) -> (b -> a -> b)-> b -> [a] -> [
    a]
takeWhileAccumulating predicate accumulate init list = map fst . L.
    takeWhile (predicate . snd) . L.zip list . tail . L.scanl
    accumulate init $ list


fromReversedList :: [a] -> Seq a
fromReversedList =  L.foldl (flip (<|)) S.empty

seqDelete :: (Eq a) => a -> Seq a -> Seq a
seqDelete x xs = fromMaybe xs deleted
  where deleted     = (joinOthers . splitSeqAt) ‘fmap‘ pos
        splitSeqAt  = flip Data.Sequence.splitAt xs
        joinOthers  = uncurry ((. Data.Sequence.drop 1) . (><))
        pos         = L.elemIndex x . toList $ xs
```

Listing 5: Simulation/Loggers.hs

```
module Simulation.Loggers(Loggable,  runLoggers, echoStates,
    echoMessages, logStates, logMessages, none, output) where
import Simulation.Utils
import Simulation.Simulation
import Simulation.LimitOrderBook
import Simulation.Trade
import Simulation.Order hiding (Market)
import Simulation.OrderResponse
import Simulation.Market
import Text.CSV
import Control.Monad
import Control.Arrow
import Data.List
import System.Directory
import Control.Monad.Error

class Loggable a where
  output :: a -> [(Field, Field)]

data (Loggable s, Loggable m) => Logger s m = Logger {
  setup :: s -> [m] -> IO (),
  run :: s -> [m] -> IO ()
}

none = ""

keys :: (Loggable a) => a -> Record
keys = map fst . output

```

```
values :: Loggable a => a -> Record
values = map snd . output

runLoggers :: (Loggable s, Loggable m, Message m) => [Logger s m] -> SimResult m s -> IO (SimResult m s)
runLoggers loggers result = do
  let pairs = zip (states result) (groupBy sameTime . messages $ result)
  distM_ (map (uncurry . setup) loggers) $ head pairs
  mapM_ (distM_ $ map (uncurry . run) loggers) $ pairs
  return result

echoStates :: (Loggable s, Loggable m) => Logger s m
echoStates = Logger noSetup runEchoStates

echoMessages :: (Loggable s, Loggable m) => Logger s m
echoMessages = Logger noSetup runEchoMessages

logStates :: (Loggable s, Loggable m) => FilePath -> Logger s m
logStates path = Logger (setupLogStates path) (runLogStates path)

logMessages :: (Loggable s, Loggable m) => FilePath -> Logger s m
logMessages path = Logger (setupLogMessages path) (runLogMessages path)


runEchoStates :: (Loggable s, Loggable m) =>  s -> [m] -> IO ()
runEchoStates state messages = do
  putStrLn . ("State: " ++) . show . values $ state

runEchoMessages :: (Loggable s, Loggable m) =>  s -> [m] -> IO ()
runEchoMessages state messages = do
  mapM_ (putStrLn . ("    Message: " ++) . show . values) $ messages

runLogStates :: (Loggable s, Loggable m) => FilePath -> s -> [m] -> IO ()
runLogStates filename state messages = do
  let csv  = printCSV . return . values $ state
  appendFile filename $ csv ++ "\n"

runLogMessages :: (Loggable s, Loggable m) => FilePath -> s -> [m] -> IO ()
runLogMessages filename state messages = do
  let csv = printCSV . map values $ messages
  appendFile filename csv


setupLogStates :: (Loggable s, Loggable m) => FilePath -> s -> [m] -> IO ()
setupLogStates  filename state messages = do
  exists <- doesFileExist filename
  if exists then
    removeFile filename
    else return ()
  appendFile filename $ printCSV (return . keys $ state) ++ "\n"
  return ()
setupLogMessages :: (Loggable s, Loggable m) => FilePath -> s -> [m] -> IO ()
setupLogMessages filename state messages = do
  exists <- doesFileExist filename
  if exists then
    removeFile filename
```

```
        else return ()
      appendFile filename $ printCSV (return . keys . head $ messages) ++
           "\n"
    noSetup :: (Loggable s, Loggable m) => s -> [m] -> IO ()
    noSetup state messages = return ()

    sameTime :: (Message m1, Message m2) => m1 -> m2 -> Bool
    sameTime m1 m2 = messageTime m1 == messageTime m2
```

Listing 6: Simulation/Market.hs

```
{-# LANGUAGE ExistentialQuantification #-}
module Simulation.Market where
  import Data.List
  import Simulation.Types
  import Simulation.Simulation
  import Simulation.LimitOrderBook
  import Simulation.Order hiding (Market)
  import Simulation.OrderResponse
  import Simulation.Trade
  import Control.Monad.Writer
  import Control.Monad.State
  import Data.Map (Map, (!))
  import qualified Data.Map as M

  data Market = Market {
    time :: Time,
    value :: Value,
    book :: LOB,
    lastBestBid :: Price,
    lastBestOffer :: Price,
    lastNumOrders :: Int
  }

  data MarketOutlook = CrossedRising | CrossedStable | CrossedFalling |
       Rising | Stable | Falling deriving (Eq, Show)


  data MarketMessage = Response Time OrderResponse | forall a b . (
      MarketSide a, OrderType b) => Message Time (Order a b)

  instance Message MarketMessage where
    messageTime      (Response t _) = t
    messageTime      (Message  t _) = t

  newtype TraderState = TraderState {
    inventoryLevel :: InventoryLevel
  }

  makeMarket :: Value -> LOB -> Market
  makeMarket v b = Market 0 v b (getPrice v 0) (getPrice v 0) 0

  updateInventory :: TraderState -> InventoryLevel -> TraderState
  updateInventory (TraderState i) i’ = TraderState $ i + i’

  data Sentiment = Calm | Choppy | Ramp | Toxic
  newtype Value = Value { getPrice :: Time -> Price }

  currentValue :: Market -> Price
  currentValue market = getPrice (value market) (time market)

```

```haskell
underlyingValue :: Sentiment -> Price -> Int -> Value
underlyingValue sentiment initial ticks = Value (function sentiment initial)
  where function Calm   price tick = price
        function Choppy price tick = max (floor $ (sines !! tick + 1) * fromIntegral price) 0
        function Ramp   price tick = max (floor $ (rampDown !! tick) * fromIntegral price) 0
        function Toxic  price tick = if tick < 40 then price else 5
        sines     = cycle [sin x | x <- [0.0, (6*pi/fromIntegral ticks) .. 2*pi]]
        rampDown  = [1.0, 1.0-(1/fromIntegral ticks) ..]

updateMarket :: Market -> [MarketMessage] -> SimState TraderState MarketMessage Market
updateMarket last messages = do
  let tick                = time last
  let book'               = book last
  let (book'', responses) = placeOrders tick book' messages
  tell $ map (Response tick) responses
  agentState <- get
  put $ M.mapWithKey (updateStateForAgent responses) agentState
  return $  Market (tick + 1) (value last) book'' (bestBid book') (bestOffer book') (numOrders book')

outlook :: Market -> MarketOutlook
outlook market | bb >= bo  && bo >= cv  = CrossedRising
               | bb >= cv  && cv >= bo = CrossedStable
               | cv >= bb  && bb >= bo  = CrossedFalling
               | bo >= bb  && bb >= cv  = Falling
               | bo >= cv  && cv >= bb = Stable
               | cv >= bo  && bo >= bb  = Rising
               where bb = bestBid $ book market
                     bo = bestOffer $ book market
                     cv = currentValue market

updateStateForAgent :: [OrderResponse] -> AgentID -> TraderState -> TraderState
updateStateForAgent responses id previous = foldl ((. inventoryDifference) . updateInventory) previous . filter (elem id . forTraders) $ responses
  where inventoryDifference (PenaltyResponse _ _) = 0
        inventoryDifference (TradeResponse t) | buySideTraderID t == id = tradedSize t
                                              | otherwise = -(tradedSize t)

placeOrders :: Time -> LOB -> [MarketMessage] -> (LOB, [OrderResponse])
placeOrders time book messages = placeOrders' time (book, []) messages
  where placeOrders' _ (book, responses) []       = (book, responses)
        placeOrders' t (book, responses) (m:ms)   = placeOrders' t (book', responses ++ responses') ms
          where (book', responses') =  placeOrder' t book m
                placeOrder' t b (Message time o) =  placeOrder t b o
```

Listing 7: Simulation/Market/LoggableInstances.hs

```haskell
{-# LANGUAGE GADTs #-}
```

```
module Simulation.Market.LoggableInstances where

  import Simulation.Loggers
  import Simulation.Market
  import Simulation.LimitOrderBook
  import Simulation.Trade
  import Simulation.Order hiding (Market)
  import Simulation.OrderResponse

  instance Loggable Market where
    output m = [
            ("time",                    show . time $ m),
            ("outlook",                 show . outlook $ m),
            ("currentValue",            show . currentValue $ m),
            ("bestBid",                 show . bestBid . book $ m),
            ("bestOffer",               show . bestOffer . book $ m),
            ("buySideLiquidity",        show . buySideLiquidity . book $ m),
            ("sellSideLiquitity",       show . sellSideLiquidity . book $ m),
            ("buySideDepth",            show . buySideDepth . book $ m),
            ("sellSideDepth",           show . sellSideDepth . book $ m),
            ("buySideDepthNearTop",     show . flip buySideDepthNearTop 0.05 . book $ m),
            ("sellSideDepthNearTop",    show . flip sellSideDepthNearTop 0.05 . book $ m),
            ("midPrice",                show . midPrice . book $ m),
            ("lastTradedPrice",         show . lastTradedPrice . book $ m)]

  instance Loggable MarketMessage where
    output m = zip labels values
      where labels = ["time", "type", "buySideTrader", "sellSideTrader", "size", "price", "penalty"]
            values :: [String]
            values = case m of
              (Response time (TradeResponse t))              -> [show time, "Trade",   buySideTraderID t, sellSideTraderID t, show $ tradedSize t, show $ tradedPrice t, none]
              (Response time (PenaltyResponse id p))         -> [show time, "Penalty", id,                none,              none,                none,                show p]
              (Message  time (BuyOrder id quantity))         -> [show time, "Buy",     id,                none,              show quantity,       none,                none]
              (Message  time (SellOrder id quantity))        -> [show time, "Sell",    id,                none,              show quantity,       none,                none]
              (Message  time (BidOrder id price quantity))   -> [show time, "Bid",     id,                none,              show quantity,       show price,          none]
              (Message  time (OfferOrder id price quantity)) -> [show time, "Offer",   id,                none,              show quantity,       show price,          none]
```

Listing 8: Simulation/Order.hs

```haskell
{-# LANGUAGE GADTs, EmptyDataDecls, FlexibleInstances #-}

module Simulation.Order where
  import Simulation.Types
  import Data.Maybe

  data Buy
  data Sell
  data Market
  data Limit

  class MarketSide a
  instance MarketSide Buy
  instance MarketSide Sell

  class OrderType a
  instance OrderType Market
  instance OrderType Limit

  data Order side orderType where
    BuyOrder    :: TraderID -> Size           -> Order Buy Market
    SellOrder   :: TraderID -> Size           -> Order Sell Market
    BidOrder    :: TraderID -> Price -> Size  -> Order Buy Limit
    OfferOrder  :: TraderID -> Price -> Size  -> Order Sell Limit

  instance Show (Order Buy Market) where
    show (BuyOrder t1 s1) = "BuyOrder " ++ show t1 ++  " " ++ show s1

  instance Show (Order Sell Market) where
    show (SellOrder t1 s1) = "SellOrder " ++ show t1 ++ " " ++ show s1

  instance Show (Order Sell Limit) where
    show (OfferOrder t1 p1 s1) = "OfferOrder " ++ show t1 ++ " " ++
        show p1 ++ " "++ show s1

  instance Show (Order Buy Limit) where
    show (BidOrder t1 p1 s1) = "BidOrder " ++ show t1 ++ " " ++ show p1
         ++ " " ++ show s1

  instance Eq (Order Buy Market) where
    (BuyOrder t1 s1) == (BuyOrder t2 s2) = t1 == t2 && s1 == s2

  instance Eq (Order Sell Market) where
    (SellOrder t1 s1) == (SellOrder t2 s2) = t1 == t2 && s1 == s2

  instance Eq (Order Buy Limit) where
    (BidOrder t1 p1 s1) == (BidOrder t2 p2 s2) = t1 == t2 && s1 == s2
        && p1 == p2

  instance Eq (Order Sell Limit) where
    (OfferOrder t1 p1 s1) == (OfferOrder t2 p2 s2) = t1 == t2 && p1 ==
        p2 && s1 == s2

  instance Ord (Order Buy Limit) where
    (BidOrder t1 p1 _) `compare` (BidOrder t2 p2 _) = p1 `compare` p2

  instance Ord (Order Sell Limit) where
    (OfferOrder t1 p1 _) `compare` (OfferOrder t2 p2 _) = p2 `compare`
        p1

```

```
  type BuyOrder = Order Buy Market
  type SellOrder = Order Sell Market
  type OfferOrder = Order Sell Limit
  type BidOrder = Order Buy Limit

  buys = BuyOrder
  sells = SellOrder

  bids = BidOrder
  offers = OfferOrder

  for :: MarketSide s => (Size -> Order s Limit) -> Size -> Order s Limit
  f ‘for‘ a = f a

  traderID :: (Order s t) -> TraderID
  traderID (BuyOrder    t _)    = t
  traderID (SellOrder   t _)    = t
  traderID (BidOrder    t _ _)  = t
  traderID (OfferOrder  t _ _)  = t


  size :: (Order s t) -> Size
  size (BuyOrder    _ s)    = s
  size (SellOrder   _ s)    = s
  size (BidOrder    _ _ s)  = s
  size (OfferOrder  _ _ s)  = s

  price :: (Order s Limit) -> Price
  price (BidOrder   _ p _) = p
  price (OfferOrder _ p _) = p

  updateSize :: Size -> (Order s Limit) -> (Order s Limit)
  updateSize s (BidOrder    t _ p) = BidOrder t s p
  updateSize s (OfferOrder  t _ p) = OfferOrder t s p
```

Listing 9: Simulation/OrderResponse.hs

```
module Simulation.OrderResponse where
  import Simulation.Types
  import Simulation.Trade
  import Control.Applicative

  data OrderResponse = TradeResponse Trade | PenaltyResponse TraderID Penalty

  forTraders :: OrderResponse -> [TraderID]
  forTraders (TradeResponse trade)      = [buySideTraderID, sellSideTraderID] <*> (return trade)
  forTraders (PenaltyResponse trader _) = [trader]
```

Listing 10: Simulation/Simulation.hs

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Simulation.Simulation (AgentID, Message(messageTime), SimState, Agent(Agent), Simulation(Simulation), SimResult(states, messages), runSim) where
  import Simulation.Utils
  import Control.Applicative
```

```haskell
import Control.Arrow
import Control.Monad
import Control.Monad.State
import Control.Monad.Writer
import Data.Map (Map)
import qualified Data.Map as M

type AgentID = String

class Message m where
  messageTime :: m -> Int

newtype (Message m) => SimState a m s = SimState { unState :: StateT (Map AgentID a) (WriterT [m] IO) s } deriving (Monad, MonadIO, MonadWriter [m], MonadState (Map AgentID a), Functor)

data (Message m) => Agent a m s = Agent { agentID :: AgentID,
                           behaviour :: s -> SimState a m (),
                           initialAgentState :: a
                         }

data (Message m) => Simulation a m s = Simulation { agents :: [Agent a m s],
                                     initialState :: s,
                                     reduce :: s -> [m] -> SimState a m s
                                   }

data (Message m) => SimResult m s = SimResult {states :: [s], messages :: [m]}

runSim :: (Message m) => Int -> Simulation a m s -> IO (SimResult m s)
runSim iterations sim = makeResult (simStep sim) (initialState sim)
  where makeResult step init  = liftM (uncurry SimResult) . runWriterT . flip evalStateT agentStates . unState $ states
          where agentStates           = M.fromList . map (agentID &&& initialAgentState) . agents $ sim
                states                = iterateM iterations step init

simStep :: (Message m) => Simulation a m s -> s -> SimState a m s
simStep sim state = do
  let functions = map behaviour $ agents sim
  (_, messages) <- listen $ return state >>= distM_ functions
  (reduce sim) state messages
```

Listing 11: Simulation/Trade.hs

```haskell
{-# LANGUAGE GADTs #-}
module Simulation.Trade (Trade(BuyTrade, SellTrade, CrossedTrade), makeTrade, buySideTraderID, sellSideTraderID, tradedSize, tradedPrice) where
  import Simulation.Types
  import Simulation.Order

  data Trade =
    BuyTrade     (Order Buy Market) (Order Sell Limit)  |
    SellTrade    (Order Buy Limit)  (Order Sell Market) |
    CrossedTrade (Order Buy Limit)  (Order Sell Limit)

```

```
makeTrade :: (OrderType a, OrderType b) => (Order Buy a) -> (Order Sell b) -> Trade
makeTrade buyOrder sellOrder = case (buyOrder,sellOrder) of
  ((BuyOrder _ _  ), (OfferOrder _ _ _)) -> BuyTrade     buyOrder sellOrder
  ((BidOrder _ _ _), (SellOrder _ _   )) -> SellTrade    buyOrder sellOrder
  ((BidOrder _ _ _), (OfferOrder _ _ _)) -> CrossedTrade buyOrder sellOrder

buySideTraderID :: Trade -> TraderID
buySideTraderID (BuyTrade     (BuyOrder t _)     (OfferOrder _ _ _)) = t
buySideTraderID (SellTrade    (BidOrder t _ _)   (SellOrder  _ _)) = t
buySideTraderID (CrossedTrade (BidOrder t _ _)   (OfferOrder _ _ _)) = t

sellSideTraderID :: Trade -> TraderID
sellSideTraderID (BuyTrade     (BuyOrder _ _)     (OfferOrder t _ _)) = t
sellSideTraderID (SellTrade    (BidOrder _ _ _)   (SellOrder  t _)) = t
sellSideTraderID (CrossedTrade (BidOrder _ _ _)   (OfferOrder t _ _)) = t

tradedSize :: Trade -> Size
tradedSize trade = uncurry min $ sizes
  where sizes :: (Size, Size)
        sizes = case trade of
          (BuyTrade     (BuyOrder _ bs)   (OfferOrder _ _ ss)) -> (bs, ss)
          (SellTrade    (BidOrder _ _ bs) (SellOrder  _ ss))   -> (bs, ss)
          (CrossedTrade (BidOrder _ _ bs) (OfferOrder _ _ ss)) -> (bs, ss)

tradedPrice :: Trade -> Price
tradedPrice (BuyTrade     (BuyOrder _ _)    (OfferOrder _ p _))  = p
tradedPrice (SellTrade    (BidOrder _ p _)  (SellOrder  _ _))    = p
tradedPrice (CrossedTrade (BidOrder _ bp _)  (OfferOrder _ sp _)) = (bp + sp) ‘div‘ 2
```

Listing 12: Simulation/Traders.hs

```
module Simulation.Traders (Trader, makeTraders) where
  import Simulation.Types
  import Simulation.Traders.Types
  import Simulation.Traders.Pricing
  import Simulation.Utils
  import Simulation.Constants
  import Simulation.Order hiding (Market, traderID)
  import Simulation.Market
  import Simulation.LimitOrderBook hiding (placeOrder)
  import Simulation.Trade
  import Simulation.OrderResponse
  import Simulation.Simulation
  import Data.Map (Map, (!))
  import Control.Monad.Writer
  import Control.Monad.State

```

```
makeTraders :: [(Int,Trader)] -> [Agent TraderState MarketMessage Market]
makeTraders pairs = pairs >>= uncurry fromPair
  where fromPair n trader = map (flip makeAgent trader) [1..n]

makeAgent :: Int -> Trader -> Agent TraderState MarketMessage Market
makeAgent n trader = Agent tid function initialState
  where tid = (traderID trader ++ "-" ++ show n)
        initialState = TraderState $ initialInventory trader
        function market = do
          (TraderState inventoryLevel) <- (flip (!) $ tid) ‘fmap‘ get
          let targetInv       = inventoryLevel + supplyDemand trader * time market
          let oSize           = orderSize trader market targetInv
          let oType           = orderType oSize (orderSizeLimit trader)
          let pricingStrategy = if oSize > (orderSizeLimit trader) then aggressive else neutral
          let oPrice          = pricingStrategy oType trader market
          placeOrder (time market) oType (abs oSize) oPrice tid

placeOrder :: Time -> OrderTypeName -> Size -> Maybe Price -> TraderID -> SimState TraderState MarketMessage ()
placeOrder time t s p id = tell [makeOrder t s p id]
  where makeOrder Bid   oSize (Just oPrice) id = Message time $ id ‘bids‘ oPrice ‘for‘ oSize
        makeOrder Offer oSize (Just oPrice) id = Message time $ id ‘offers‘ oPrice ‘for‘ oSize
        makeOrder Buy   oSize _             id = Message time $ id ‘buys‘ oSize
        makeOrder Sell  oSize _             id = Message time $ id ‘sells‘ oSize

orderSize :: Trader -> Market -> Size -> Size
orderSize trader market target                 = floor $ priceChangeF priceChange * orderImbalanceF imbalance * fromIntegral (inventoryFunction trader trader (currentValue market) (midPrice . book  $ market) target)
  where priceChangeF                           = volatilityFunction trader
        orderImbalanceF                        = imbalanceFunction trader
        priceChange                            = 2 * (bidMovement + offerMovement)
        imbalance                              = ssDepth - bsDepth
        bsDepth                                = buySideDepthNearTop book’ topOfBookThreshold
        ssDepth                                = sellSideDepthNearTop book’ topOfBookThreshold
        bidMovement   | lastBestBid market > 0 = fromIntegral (bestBid book’ - lastBestBid market) / fromIntegral (lastBestBid market)
                      | otherwise =  0.05
        offerMovement | lastBestOffer market > 0 = fromIntegral (bestOffer book’ - lastBestOffer market) / fromIntegral (lastBestOffer market)
                      | otherwise = 0.05
        book’               = book market

orderType :: Size -> Size -> OrderTypeName
orderType orderSize sizeLimit | orderSize < 0 && abs orderSize  >= sizeLimit  = Sell
```

```
                                        | orderSize < 0
                                                                                                 =
                                            Offer
                                        | orderSize >= 0 && abs orderSize >=
                                            sizeLimit  = Buy
                                        | otherwise

                                            = Bid
```

Listing 13: Simulation/Traders/InventoryFunctions.hs

```
module Simulation.Traders.InventoryFunctions(intermediaryInventory,
    hfInventory, fundamentalBuyerInventory, fundamentalSellerInventory,
     opportunisticTraderInventory, smallTraderInventory) where

  import Simulation.Types
  import Simulation.Utils
  import Simulation.Traders.Types
  intermediaryInventory :: InventoryFunction

  intermediaryInventory trader value price inventory = bounded (-2000)
      amount 2000
    where amount = floor $ 0.2 * (-(tan (y/700)))* fromIntegral (
        targetOrderSize trader)
          y      = inventoryWeight trader value price inventory

  hfInventory :: InventoryFunction
  hfInventory trader value price inventory = bounded (-2000) amount
      2000
    where amount = floor $ 0.2* (-(tan (y/700)))* fromIntegral (
        targetOrderSize trader)
          y      = inventoryWeight trader value price inventory

  fundamentalBuyerInventory :: InventoryFunction
  fundamentalBuyerInventory trader value price inventory = bounded
      (-2000) amount 2000
    where amount = floor $ exp (y/150) - fromIntegral (targetOrderSize
        trader)
          y      = inventoryWeight trader value price inventory

  fundamentalSellerInventory :: InventoryFunction
  fundamentalSellerInventory trader value price inventory = bounded
      (-2000) amount 2000
    where amount = floor $ exp (-y/150) - fromIntegral (targetOrderSize
         trader)
          y      = inventoryWeight trader value price inventory

  opportunisticTraderInventory :: InventoryFunction
  opportunisticTraderInventory trader value price inventory = bounded
      (-250) amount 250
    where amount = floor $ (- tan (y/700)) * fromIntegral (
        targetOrderSize trader)
          y      = inventoryWeight trader value price inventory

  smallTraderInventory :: InventoryFunction
  smallTraderInventory trader value price inventory = bounded (-250)
      amount 250
    where amount = floor$ (- tan (y/700)) * fromIntegral (
        targetOrderSize trader)
          y      = inventoryWeight trader value price inventory

```

```
inventoryWeight trader value price inventory = fromIntegral inventory
    - fromIntegral (targetInventory trader) * ((1+) . sigmoid .
    fromIntegral $ value - price)
```

Listing 14: Simulation/Traders/Pricing.hs

```
module Simulation.Traders.Pricing where
  import Simulation.Types
  import Simulation.Traders.Types
  import Simulation.Constants
  import Simulation.Market
  import Simulation.LimitOrderBook

  aggressive :: OrderTypeName -> Trader -> Market -> Maybe Price
  aggressive orderType trader market = case orderType of
    Bid   -> Just $ aggressiveBuy trader market
    Offer -> Just $ aggressiveSell trader market
    Buy   -> Nothing
    Sell  -> Nothing

  neutral :: OrderTypeName -> Trader -> Market -> Maybe Price
  neutral orderType trader market = case orderType of
    Bid   -> Just $ neutralBuy trader market
    Offer -> Just $ neutralSell trader market
    Buy   -> Nothing
    Sell  -> Nothing


  neutralBuy :: Trader -> Market -> Price
  neutralBuy trader market
    | outlook' `elem` [CrossedStable, CrossedRising, Rising] = bo
    | outlook' == Stable                                     = floor $
      fromIntegral bb + 0.5 * sensitivity
    | otherwise                                              = cv
    where outlook'     = outlook market
          bo           = bestOffer $ book market
          bb           = bestBid $ book market
          cv           = currentValue market
          sensitivity  = volatilityFunction trader . fromIntegral $
              buySideDepthNearTop (book market) topOfBookThreshold


  aggressiveBuy :: Trader -> Market -> Price
  aggressiveBuy trader market
    | outlook' `elem` [CrossedFalling, CrossedStable, Rising] = bo
    | outlook' == CrossedRising                               = bb
    | outlook' == Falling                                     = floor .
       (fromIntegral cv -) . abs $ sensitivity * fromIntegral (bb -
      cv)
    | outlook' == Stable                                      = floor .
       (fromIntegral bb +) $ sensitivity * fromIntegral (bb - cv)
    where outlook'    = outlook market
          bo          = bestOffer $ book market
          bb          = bestBid $ book market
          cv          = currentValue market
          sensitivity = volatilityFunction trader . fromIntegral $
              buySideDepthNearTop (book market) topOfBookThreshold


  neutralSell :: Trader -> Market -> Price
  neutralSell trader market
```

```haskell
    | outlook' `elem` [CrossedFalling, CrossedStable, Falling] = bb
    | outlook' `elem` [Rising, CrossedRising]                 = cv
    | otherwise                                                = floor
        $ fromIntegral bo - 0.5 * sensitivity
    where  outlook'    = outlook market
           bo          = bestOffer $ book market
           bb          = bestBid $ book market
           cv          = currentValue market
           sp          = bo - bb
           sensitivity = volatilityFunction trader . fromIntegral $
               sellSideDepthNearTop (book market) topOfBookThreshold

  aggressiveSell :: Trader -> Market -> Price
  aggressiveSell trader market
    | outlook' == CrossedFalling               = bo
    | outlook' `elem` [CrossedStable, Falling]  = bb
    | outlook' == CrossedRising                = cv
    | outlook' == Rising                       = abs . (cv +) . floor
        $ abs (sensitivity * fromIntegral (cv - sp))
    | outlook' == Stable                       = abs . (bo -) . floor
        $ (sensitivity * fromIntegral sp)
    where  outlook'     = outlook market
           bo           = bestOffer $ book market
           bb           = bestBid $ book market
           cv           = currentValue market
           sp           = bo - bb
           sensitivity  = volatilityFunction trader . fromIntegral $
               sellSideDepthNearTop (book market) topOfBookThreshold
```

Listing 15: Simulation/Traders/SensitivityFunctions.hs

```haskell
module Simulation.Traders.SensitivityFunctions where
  import Simulation.Types
  import Simulation.Utils
  import Simulation.Constants

  lowImbalanceSensitivity :: ImbalanceFunction
  lowImbalanceSensitivity = const 1

  mediumImbalanceSensitivity :: ImbalanceFunction
  mediumImbalanceSensitivity = (0.5-) . sigmoid . fromIntegral

  highImbalanceSensitivity :: ImbalanceFunction
  highImbalanceSensitivity = min highSensitivityLimit . exp .
      fromIntegral

  lowVolatilitySensitivity :: VolatilityFunction
  lowVolatilitySensitivity = const 1

  mediumVolatilitySensitivity :: VolatilityFunction
  mediumVolatilitySensitivity = (0.5-) . sigmoid

  highVolatilitySensitivity :: VolatilityFunction
  highVolatilitySensitivity = min highSensitivityLimit . exp
```

Listing 16: Simulation/Traders/Types.hs

```haskell
module Simulation.Traders.Types where
  import Simulation.Types
```

```
newtype HasInitialInventory = II Size
newtype HasTargetInventory = TI Size
newtype HasTargetOrderSize = TOS Size
newtype HasDemandBias = DB Int
newtype HasOrderSizeLimit = OSL Size
newtype HasInventoryFunction = IF InventoryFunction

data NoInitialInventory = NoII
data NoTargetInventory = NoTI
data NoTargetOrderSize = NoTOS
data NoDemandBias = NoDB
data NoOrderSizeLimit = NoOSL
data NoInventoryFunction = NoIF

class InitialInventory a
instance InitialInventory HasInitialInventory
instance InitialInventory NoInitialInventory

class TargetInventory a
instance TargetInventory HasTargetInventory
instance TargetInventory NoTargetInventory

class TargetOrderSize a
instance TargetOrderSize HasTargetOrderSize
instance TargetOrderSize NoTargetOrderSize

class DemandBias a
instance DemandBias HasDemandBias
instance DemandBias NoDemandBias

class OrderSizeLimit a
instance OrderSizeLimit HasOrderSizeLimit
instance OrderSizeLimit NoOrderSizeLimit

class InventoryFunc a
instance InventoryFunc HasInventoryFunction
instance InventoryFunc NoInventoryFunction

data (InitialInventory ii, TargetInventory ti, TargetOrderSize tos,
    DemandBias db, OrderSizeLimit osl, InventoryFunc inf) =>
    TraderType ii ti tos db osl inf = Trader String ImbalanceFunction
     VolatilityFunction ii ti tos db osl inf

type Trader = TraderType HasInitialInventory HasTargetInventory
    HasTargetOrderSize HasDemandBias HasOrderSizeLimit
    HasInventoryFunction

data OrderTypeName = Buy | Sell | Bid | Offer deriving (Eq, Show)
type InventoryFunction  = Trader -> Price -> Price -> Size -> Size

traderType :: String -> ImbalanceFunction -> VolatilityFunction ->
    TraderType NoInitialInventory NoTargetInventory NoTargetOrderSize
     NoDemandBias NoOrderSizeLimit NoInventoryFunction
traderType name imbalance volatility = Trader name imbalance
    volatility NoII NoTI NoTOS NoDB NoOSL NoIF

withInitialInventory :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf -> Size ->
    TraderType HasInitialInventory ti tos db osl inf
```

```
withInitialInventory (Trader tid imbf vf _ ti tos db osl invf) ii =
    Trader tid imbf vf (II ii) ti tos db osl invf

withTargetInventory :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf -> Size ->
     TraderType ii HasTargetInventory tos db osl inf
withTargetInventory (Trader tid imbf vf ii _ tos db osl invf) ti =
    Trader tid imbf vf ii (TI ti) tos db osl invf

withTargetOrderSize :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf -> Size ->
    TraderType ii ti HasTargetOrderSize db osl inf
withTargetOrderSize (Trader tid imbf vf ii ti _ db osl invf) tos =
    Trader tid imbf vf ii ti (TOS tos) db osl invf

withDemandBias :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf -> Int ->
    TraderType ii ti tos HasDemandBias osl inf
withDemandBias (Trader tid imbf vf ii ti tos _ osl invf) db = Trader
    tid imbf vf ii ti tos (DB db) osl invf

withOrderSizeLimit :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf -> Size ->
    TraderType ii ti tos db HasOrderSizeLimit inf
withOrderSizeLimit (Trader tid imbf vf ii ti tos db _ invf) osl =
    Trader tid imbf vf ii ti tos db (OSL osl) invf

withInventoryFunction :: (InitialInventory ii, TargetInventory ti,
    TargetOrderSize tos, DemandBias db, OrderSizeLimit osl,
    InventoryFunc inf) => TraderType ii ti tos db osl inf ->
    InventoryFunction -> TraderType ii ti tos db osl
    HasInventoryFunction
withInventoryFunction (Trader tid imbf vf ii ti tos db osl _) invf =
    Trader tid imbf vf ii ti tos db osl (IF invf)

traderID :: Trader -> TraderID
traderID (Trader tid _ _ _ _ _ _ _ _) = tid

imbalanceFunction :: Trader -> ImbalanceFunction
imbalanceFunction (Trader _ imbf _ _ _ _ _ _ _) = imbf

volatilityFunction ::  Trader -> VolatilityFunction
volatilityFunction (Trader _ _ vf _ _ _ _ _ _) = vf

initialInventory :: Trader -> InventoryLevel
initialInventory (Trader _ _ _ (II ii) _ _ _ _ _) = ii

targetInventory :: Trader -> InventoryLevel
targetInventory (Trader _ _ _ _ (TI ti) _ _ _ _) = ti

targetOrderSize :: Trader -> Size
targetOrderSize (Trader _ _ _ _ _ (TOS tos) _ _ _) = tos

supplyDemand :: Trader -> Int
supplyDemand (Trader _ _ _ _ _ _ (DB db) _ _) = db

orderSizeLimit :: Trader -> Size
orderSizeLimit (Trader _ _ _ _ _ _ _ (OSL osl) _) = osl
```

```haskell

inventoryFunction :: Trader -> InventoryFunction
inventoryFunction (Trader _ _ _ _ _ _ _ _ (IF inf)) = inf
```

Listing 17: Simulation/Types.hs

```haskell
module Simulation.Types where
  type Price = Int
  type Size = Int
  type Time = Int
  type TraderID = String
  type Penalty = Int
  type InventoryLevel = Int

  type VolatilityFunction = Double -> Double
  type ImbalanceFunction  = Int -> Double
```

Listing 18: Simulation/Utils.hs

```haskell
module Simulation.Utils where
  import Control.Monad

  bounded :: (Ord a) => a -> a -> a -> a
  bounded = (min .) . max

  sigmoid x = 1 / (1 + exp(-x))

  distM_ :: (Monad m) => [a -> m b] -> a -> m ()
  distM_ fs x = sequence_ . map ($x) $ fs

  iterateM :: Monad m => Int -> (a -> m a) -> a -> m [a]
  iterateM (-1)  _ _ = return []
  iterateM  n    f a = (a:) `liftM` (f a >>= iterateM (n-1) f)
```

Acknowledgements

The completion of this project would not have been possible without the help and support of many people, including but not limited to: Christopher Clack, who acted as my supervisor at UCL, and the denizens of the haskell-beginners and haskell-cafe mailing lists as well as the #haskell IRC channel who offered a great deal of advice. In particular: Brent Yorgey, Keegan McAllister, Cale Gibbard, Jules Bean, Dylan Lukes, and many known to me only by their IRC nicknames: ”merijn”, ”ski”, ”acowley”, ”ivanm”, ”jmcarthur”, ”rostayob”, ”danr”, ”Saizan”, ”Botje” and ”gienah”. Daniel Shore and Henry Garner of Likely Ltd. were very accommodating of my work on this project, allowing me to take time off from my professional work to complete it, Henry (along with Matthew Willson) also acted as an invaluable sounding board for ideas developed during the course of this project. Finally, Vicky Holland provided an unmeasurable amount of support and patience throughout the entire process. Thank you all.

Bibliography

Agha, G. (1985), ‘Actors: a model of concurrent computation in distributed systems’, *Information Systems* .

Brown, N. (2008), ‘Communicating Haskell Processes: Composable explicit concurrency using monads’, *Communicating Process Architectures* pp. 67–83.

Cheney, J. & Hinze, R. (2003), ‘First-class phantom types’, pp. 1–32.

Clack, C. D. (2011), System Instability in the Financial Markets.

Darley, V. & Outkin, A. (2007), *A NASDAQ market simulation: insights on a major market from the science of complex adaptive systems*, World Scientific Pub Co Inc.

Evans, E. (2003), *Domain-Driven Design: Tackling complexity in the heart of software*, Vol. 70, Addison-Wesley Professional.

Farmer, J. & Foley, D. (2009), ‘The economy needs agent-based modelling’, *Nature* **460**(7256), 685–686.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design patterns: elements of reusable object-oriented software*, Vol. 206, Addison-Wesley Professional.

Ghosh, D. (2010), *DSLs in Action*, Manning.

Ghosh, D. (2011), ‘DSL for the uninitiated’, *ACM Queue* pp. 1–11.

Hoare, C. (1978), ‘Communicating sequential processes’, *Communications of the ACM* **21**(8).

Holland, J. & Miller, J. (1991), ‘Artificial adaptive agents in economic theory’, *The American Economic Review* **81**(2), 365–370.

Hudak, P. (1996), ‘Building domain-specific embedded languages’, *ACM Computing Surveys* **28**(4es), 196–es.

Hughes, J. (2005), ‘Programming with arrows’, *Advanced Functional Programming* pp. 73–129.

Iry, J. (2010), ‘Phantom Types in Haskell and Scala’.
`http://tinyurl.com/phantomtypes`

Jones, M. (1995), 'Functional programming with overloading and higher-order polymorphism', *Advanced Functional Programming* pp. 97–136.

Kirilenko, A., Kyle, A., Samadi, M. & Tuzun, T. (2010), 'The flash crash: The impact of high frequency trading on an electronic market', *Manuscript, U of Maryland* .

Laufer, K. (1996), 'Type classes with existential types', *Journal of Functional Programming* **6**(3), 485–518.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K. & Balan, G. (2005), 'MASON: A multiagent simulation environment', *Simulation* **81**(7), 517.

Macal, C. & North, M. (2010), 'Tutorial on agent-based modelling and simulation', *Journal of Simulation* **4**(3), 151–162.

Mak, S. (2007), 'Developing interacting domain specific languages', (November).

Malleson, N., Heppenstall, A. & See, L. (2010), 'Crime reduction through simulation: An agent-based model of burglary', *Computers, Environment and Urban Systems* **34**(3), 236–250.

Minar, N., Burkhart, R., Langton, C. & Askenazi, M. (1996), 'The swarm simulation system: A toolkit for building multi-agent simulations'.

Moggi, E. (1988), Computational lambda-calculus and monads, *in* 'Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on', IEEE, pp. 14–23.

O'Sullivan, B., Goerzen, J. & Stewart, D. (2008), *Real World Haskell*, O'Reilly.

Peyton Jones, S. (2003), *Haskell 98 language and libraries: the revised report*, Cambridge Univ Pr.

Peyton Jones, S. & Eber, J. (2003), 'How to write a financial contract'.

Peyton Jones, S., Hall, C., Hammond, K., Cordy, J. & Kevin, H. (1992), 'The Glasgow Haskell compiler: a technical overview', *Computing* pp. 1–9.

Van Deursen, A., Klint, P. & Visser, J. (2000), 'Domain-specific languages: An annotated bibliography', *ACM Sigplan Notices* **35**(6), 36.

Van Dyke Parunak, H., Savit, R. & Riolo, R. (1998), Agent-based modeling vs. equation-based modeling: A case study and users guide, *in* 'Multi-Agent Systems and Agent-Based Simulation', Springer, pp. 10–25.

Wadler, P. (1987), Views: A way for pattern matching to cohabit with data abstraction, *in* 'Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM, pp. 307–313.

Wadler, P. (1992), 'Comprehending monads', *Mathematical Structures in Computer Science* **2**(June 1990), 1–38.

Yorgey, B. (2009), 'The Typeclassopedia', *The Monad Reader* (13), 17–68.