Re-engineering agent based simulator with functional language

Yifei Liu

Msc Computer Science

Submission year: 2014

Supervisor: Dr. Christopher D. Clack

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

Agent based simulation is an effective tool to study and observe complex phenomena in financial market. To achieve more realistic simulation, we need to improve the performance of the simulator. In this project, we re-engineer a simulator written in functional programming language. We start with capturing requirements to clarify the tasks for this project. We compare several programming languages in speed, compatibility, resources and feasibility for this project, and Haskell turns out to be the best choice. With the clear goals, we port the simulator to Haskell, and profile it to get comprehensive view of it. Based on the profiling result, we bring out optimization solutions, such as fixing inappropriate recurrence and deploying parallelism. Through this project, the simulator's performance is significantly improved.

Table of Contents

1 Introduction

1.1 Simulator for financial markets

In order to recreate and observe the Hot Potato Effect which *is a type of financial market instability that is thought to arise from complex non-linear interaction between market-making algorithms*, Christopher Clack and Elias Court [1] developed a simulator programmed with Miranda[1]. It can perform agent-based simulation to model interaction in a financial market. Simulation is a complement of theoretical analysis. It is easier to imitate the behavior in the real world with a simulator, and discover the potential connections between "equations".

The current version of the simulator is equipped with High Frequency Trader (HFT) and Chicago Mercantile Exchange's E-Mini market models. It can perform the simulation with one exchange agent and several trader agents, and output both summary statistics of the simulation and an interaction trace.

The simulator has several limitations:

1. Poor performance.

   The simulator was developed for a very small scale of simulation with only one exchange agent and several trader agents. If we want to conduct a big scale simulation more close to real life, the simulator is not fast enough to handle it.

2. Lack of scalability.

   The simulator is programmed with Miranda. Miranda is a very simple functional programming language, initially designed for teaching. It does not support parallel and concurrent programming, it lacks of library support and optimization, and only works on UNIX as interpreter. It will be very difficult to make further development on the current simulator.

[1] 'Miranda' is a trademark of Research Software Ltd

1.2 Aim of the project, and objectives

Aim: To try to overcome the limitations of the current version of the simulator, and make it suitable from general big scale experiments. Here are the measurable objectives for this aim:

- O1: Find a more powerful language to replace Miranda.
- O2: Port the simulator to the new language and validate it by comparing the simulation result with Miranda version.
- O3: Detailed analysis on new simulator's performance and detect hotspots.
- O4: optimize the simulator and check the optimization result.

1.3 Contributions of this project

I make the following contributions:

1 A generic agent-based simulator written in Haskell with user documentation, which will help users run their experiments and further development.

2 A basic guidance of how to port Miranda program to Haskell. (Section 4.1)

3 Guidance of profiling Haskell program (both sequential and parallel).

4 Optimized simulator that is able to process much bigger simulation than the original one.

5 A good base of parallel implementation on the simulator to face future challenge.

2 Context and background

2.1 Agent based simulation of financial markets

Agent based simulation [2](also called agent based model) explains the actions and interactions of autonomous agents with a view to evaluating their influence on the entire system. It observes the behavior of agents in micro-level, with the aim of reproducing and predicting the appearance of complex phenomenon, such as Flash Crash happened in May 6th 2010 in U.S. financial market [3].

Compared to a traditional mathematical model, agent based simulation is easier to synthesize with individually simply described agents. It makes agents easier to investigate, test and debug. It can tell us the information of both agents and system.

2.2 Functional programming

Functional programming is one of programming paradigms in computer science. The fundamental operations in functional programming is evaluating expressions, in other words, applying functions to arguments. Functional programming is more like illustrating a problem with a solution. In contrast, our familiar programming languages such as C/C++ and Java, which belong to imperative programming paradigm, give commands to computer to execute a sequence of actions.

Generally, there is no *side effect* in functional program – once a function is evaluated, its value will never change. [4] This is also known as *referential transparency*. A programmer need not understand how computer works, such as memory management, and consider the flow of control. Functional languages' syntax and semantics are usually simple, so that functional programs usually have less code and is easier to get work with than imperative counterparts. Another outstanding feature of functional programming is that it follows mathematical rules to deal with expressions and values which are therefore *well suited to formal reasoning.* Hence, high-level abstraction is achievable.

2.2.1 Miranda

Miranda is a *non-strict*, *polymorphic* and *purely functional* programming language design by David Turner of University of Kent in 1983. Miranda's basic ideas develop from earlier programming

languages SASL [5] [6] and KRC [7]. "The aim of the Miranda system is to provide a modern function language, embedded in a convenient programming environment, suitable both for teaching and as a general purpose programming tool." [8]. Miranda's advantages are:

1. *Purely functional* – There is no imperative features in Miranda. *Side effect* is eliminated in Miranda. Programmer will face less burden of debugging and issues of control.
2. *Lazy evaluation* – The only evaluation mechanism in Miranda is "lazy" that the expression is not be evaluated until its value is required. Thus, non-strict functions and infinite data structures (i.e. infinite lists) can be achieved[2].
3. *Polymorphic strong typing* – every expression in Miranda has a type which can be *polymorphic* [9]. Miranda does not demand declare a type for an expression, but it will be inferred at compile time, and any type inconsistency will result in compile error. Whereas, a function can have many types due to polymorphism.
4. User defined type and abstract data type – this gives programmer more flexibility on type system and type abstraction.
5. Simple syntax and semantics – it is easy for programmer to get to work with Miranda. Miranda is a good choice for building a prototype.

As Miranda is not initially designed for commercial software development, there are several drawbacks limit Miranda's performance:

1. Lack of IO system support. There is only a simple interface with very basic functions between Miranda and UNIX.
2. Relative poor performance. Either on Miranda's compiler and language itself, there is not much effort to make the code run faster.
3. Single-threaded. In fact, there is no thread concept in Miranda. Parallel and concurrent programming is not supported.
4. Hard to profile. Miranda does not support profiling by itself, and there is no other tool to profile Miranda program. We can only get limited running information from Miranda's */count* command

[2] Non-strict functions can return answers with some of their arguments are undefined [29]

and UNIX system. Thus, optimization on Miranda program is difficult.

Miranda System runs under UNIX, and it is not open-source. It can be downloaded from Miranda website (http://miranda.org.uk/).

2.2.2 Haskell

The first version of Haskell (Haskell 1.0) was published in 1990 by the committee founded after the conference on "Functional Programming Languages and Computer Architecture (FPCA '87)" in Portland, Oregon. [10] Haskell is an advanced purely functional programming language with *strong typing*, *non-strict semantics* for general-purpose usage. The current version is Haskell 2010 [11] . Haskell is under strong influence of Miranda, and possesses most Miranda's advantages, such as *lazy evaluation*, *strong typing*, etc. But Haskell is more powerful than Miranda. Here are outstanding features of Haskell compared to Miranda:

1. *Type classes* – the most powerful feature in Haskell, initially designed for overloading but developed and generalized for much wider usages [12].
2. Haskell is a committee language [10]. Haskell never stops evolving, and its abundant library resources is still rapidly increasing.
3. Genuine support for parallel and concurrent programming.
4. Support for interaction with other language (i.e. C) – Haskell's Foreign Function Interface (FFI) [13] and other third-party libraries (i.e. *Green Card*).
5. Profiling tools provided by the compiler, giving detailed information about programs.

Haskell programs can be compiled to executable files which can be run on any computer. For programmers, Haskell Platform is created for developing and debugging Haskell programs. It contains the GHC compiler [14], the Cabal build and library system [15], and stable and widely-used collection of libraries. It can be freely downloaded from *http://www.haskell.org/platform/*.

3 Requirements and target analysis

3.1 Technical requirements

3.1.1 Analysis of the simulator in Miranda

Let us review the structure of the simulator designed by Elias Court [1]. Figure 3-1 is the structure diagram of the simulator. The top-level module "sim" drives the simulator. It retrieves the simulation result and outputs to files. The simulation runs in a recursive way. There are message communications among agents and "simstep" at each time step. The communications are handled in the method of infinite stream. That means an agent looks at the next element in the stream to get the input for the next iteration.

The simulation follows the concept of Bulk Synchronous Parallel (BSP) [16] model. That is, all agents are either (i) all calculating or (ii) all communicating. This facilitates analysis of the system. Particularly on communication, at each time step n-1 agents (trader agents) send messages to 1 agent (exchange agent, or maybe more) and then that 1 agent sends messages to n-1 agents. In fact, the Miranda implementation is entirely sequential – there is no real parallelism happening. It is easy to find out that there are two bottlenecks in the simulator. Here are candidates that are responsible for the poor performance of the simulator in Miranda:

1. The bottleneck of "simstep". All the messages will merge to "simstep" and be distributed by "simstep". The speed of "simstep" handling those messages may restrict the speed of simulation.
2. The bottleneck of exchange agent. According to the simulation model, the simulation would not move to the next step until all agents finished their message processing. The exchange agent has to process all the messages from trader agents. Generally, exchange agent will face the biggest workload. Thus, the speed of exchange agent will decide the speed of simulation.
3. The limitation of Miranda. Miranda is not an efficient language. Miranda programs are naturally slow compared to other counterparts.

We need to port the simulator to a new powerful language and optimize it afterward. As to bottlenecks, we could try to speed up themselves by low-level optimization or logically solve them by

re-design these parts.

3.1.2 Requirements for project

Here are the requirements for this project:

Must have:

- The simulator need to port to a new platform. Find a feasible target which can avoid the drawbacks of Miranda.
- Detect hot spots of ported simulator. Find out the causes of the low performance of simulator with evidence.
- Based on the hot spots detecting result, bring out optimization solutions.
- Initial level optimization.
- Measurable improvement result. Run both old and new simulators under the same condition, and compared the costs.

Should have:

- Restructure the simulator to make it more suitable for further development.

Could have:

- Initial multi-threaded level optimization, such as introducing parallel computation.
- Further upgrade for simulator. Perhaps introduce computation core implemented by low-level imperative language, such as C.

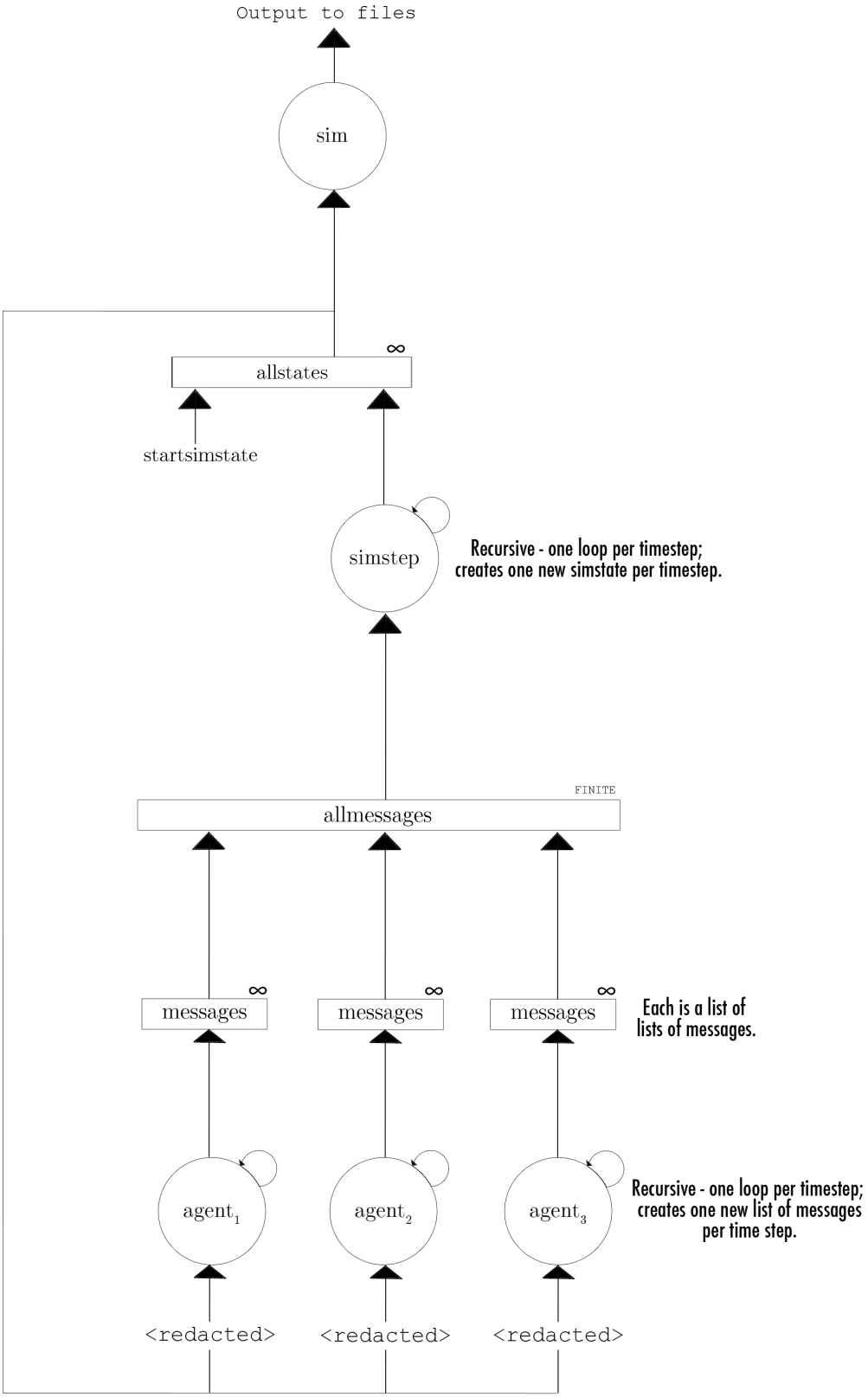


Figure 3-1 – the structure of the simulator [1]

3.2 Target analysis

3.2.1 Standard for choosing language

The original simulator is implemented by Miranda which is purely functional programming language. Imperative language gives us more freedom on optimization that we can really know how computer executes our commands so that we can change the behavior to what we want. Generally, low-level languages can run faster than high-level languages. We concern about both the efficiency of programming languages and difficulty of porting. Here are the standards for picking target language:

1. The target language should at least follow functional programming paradigm, or could be a multi-paradigm language. We do not want to completely re-build the simulator. Following the same programming paradigm will keep the original design and relieve the burden of porting. For multi-paradigm language, its imperative feature might help us on optimization.
2. The target language should have rich resources to support parallel programming. Parallel programming is the trend, and a direct solution for our case.
3. Support interaction with C or other low level imperative. There is a "could have" possibility introducing C to improve the performance of those two bottlenecks.

3.2.2 Possible candidates (F#, scala, Haskell, Ocaml)

Glancing at popular programming languages, four candidates may be suitable for the project:

1. F#

   "F# is a strongly typed, multi-paradigm programming language that encompasses functional, imperative, object-oriented programming techniques." – Wikipedia [17]. F# is developed by Microsoft and open contributors.

   Cross-platform is one of F#'s outstanding features. F# can run on Linux, Mac OS X, Android, iOS, Windows as well as HTML5 and GPUs [18]. This gives program fantastic portability. As developed by Microsoft, F# has seamless interoperability with all .NET languages and libraries, and it is open-source that helps F# evolving rapidly.

2. Scala

Scala is also a multi-paradigm programming language that encompasses object-oriented, functional, scripting programming language. Scala is proud of its scalability as its name. “Scala particularly shines when it comes to scalable server software that makes use of concurrent and synchronous processing, parallel utilization of multiple cores, and distributed processing in the cloud.” [19] Scala runs on JVM (Java Virtual Machine), so that Scala can seamlessly interoperate with Java.

3. Ocaml

   “OCaml is a general purpose industrial-strength programming language with an emphasis on expressiveness and safety.” [20] Ocaml is also capable of object-oriented, functional and imperative programming.

   Ocaml is equipped with basic features of its programming paradigms. Besides, Ocaml has its own unique advantages. One of the outstanding features is the optimization capability. With fully supported native profiling access, to compiled assembly result and sophisticated interoperation mechanism with C, Ocaml programs can be super fast, if the programmer has enough capability.

4. Haskell

   Haskell came after Miranda. Unlike other three candidates, Haskell is purely functional programming language, with heavily optimized compiler and abundant library resources. Haskell has very similar syntax and semantic with Miranda. Efficient compiler makes Haskell programs have considerable speed. Haskell’s enormous library collection ensures Haskell programs’ compatibility and scalability.

Except that F# is similar to Ocaml, these candidates are very different from each other. Initial screening is necessary before the project move to next stage.Considering the restriction of project itself and requirements, F# and Scale are ruled out at this stage. Here is the explanation:

1. Scala is the first ruled out candidate. Because Scala runs on JVM. Compared to other three candidates, Scala’s absolute speed is a concern.
2. There is no doubt F# is very powerful. F# is widely used and supported by both industrial-leading companies and open community. We do not need to worry about the

performance of F# on the simulator. But for our case, F# may be too powerful that F# may be more complex than other candidates. We may need more effort to get to work with F#, but time of the project is limited. Besides, with F#, we have to "buy in" to the Microsoft/.NET world, which might be painful. In terms of the risk and economic consideration of project, we rule out F#.

3.2.3 Haskell vs Ocaml

As what we discussed, Haskell is very different from Ocaml. It is hard to decide which one should be our target just based on the first glance. Considering our specific application, we investigate these two languages in several aspects:

1. Syntax and semantics

   This is the biggest difference between Haskell and Ocaml.

Haskell	Ocaml
Purely functional programming	Multi-paradigm programming: Object-oriented, functional and imperative.
Non-strict (lazy evaluation)	Strict, but can use lazy evaluation via explicit suspensions
Strong typing system with user-definable data type and pattern matching	Same as Haskell

Table 3-1 Syntax and semantics comparison between Haskell and Ocaml

   We should noticed that Haskell is strongly influenced by Miranda, and they have many similarities in their basic approach and in syntactic look and feel [10]. This means it is much easier to just port Miranda program to Haskell than to Ocaml.

2. Interoperation with C

   According to our initial analysis of the simulator, we might need to use imperative programming to accelerate the processing of those two bottlenecks. C which is a low-level language will be our first choice. It is important that our target language has the ability to well interoperate with C.

Haskell	Ocaml

1. Built-in support FFI. The current Haskell FFI only specifies the interaction between Haskell and C or other languages following C calling convention [13]. FFI is more like giving a basic option for user to interact with C than really providing great performance for the interoperation. Programmers are not satisfied with it, thus third-party libraries are developed. 2. Third-Party libraries. There are two available libraries – Green Card [21] and C->Haskell [22]. Unfortunately, there is not any performance report about these two libraries and we need to take a long time to figure out how to use and debug with them, even though they are open-source.	There is a sophisticated mechanism for Ocaml interoperating with C. Details are provided in Ocaml's user manual [23]. Rewriting hotspots in C is a general and ultimate optimizing method in Ocaml, because Ocaml provides friendly and well-designed mechanism for interoperation with C. In the user manual, interoperation issues such as passing value between two languages, call and callback and multithreading are described in details.

Table 3-2 Comparison of support for interoperation with C

3. Optimization capability

   Both Haskell and Ocaml have powerful profiling tools to help us detecting hotspots effectively. As functional programming language, Ocaml and Haskell follow similar optimizing way. But Ocaml has imperative feature and allows us check assembly results. That gives us deeper optimizing possibility and options.

4. Support for parallel programming

   Haskell and Ocaml both have libraries support for parallel and concurrent programming. We cannot tell which one would work better for our simulator without thoroughly experiments and tests. Since the simulator in Miranda is single-threaded, we may not be able to decide how to

treat this issue in the new platform.

5. Future development

   The future development of the simulator on Haskell and Ocaml might have very different direction. On Haskell, the simulator might be more focused on parallel programming, because there is not much space for straight-forward optimization. While on Ocaml, the simulator might evolve to Ocaml-C program to get extreme speed and efficiency.

3.2.4 Conclusion

We have discussed four possible candidates, here is the conclusion:

- Scala and F# are ruled out, considering the situation and language itself.
- Haskell and Ocaml are very different and have their own advantages. We cannot tell which one would work better for our simulator until we fully test them.
- Although we analyzed Miranda simulator, we cannot be 100% sure about the simulator will have the same problems when it is ported to a new platform. Considering this situation and the time limit of the project, **Haskell is a more feasible solution**. Because the porting workload will be minimized, and we can spend more time on analysis and solving the real problems.

4 Platform porting

We decide to port the simulator from Miranda to Haskell. Other than simply porting the code, we also improve the code structure in this project. This section provides a guidance on porting from Miranda to Haskell (section4.1), a description of how the Haskell code was restructured (section4.2) and brief verification of the efficacy of the porting (section4.3).

4.1 Transfer Miranda code to Haskell

Although Haskell has very similar syntax to Miranda, there are some issues need to be handled.

4.1.1 Expressions

There are some differences between Miranda and Haskell that we have to frequently face when porting the code.

1 Operators

Here is the table shows the major different operators in these two languages.

Miranda	Haskell	Description
~	not	Logic NOT
--	Not applicable	There is not direct list subtraction in Haskell. "--" in Haskell is the start of single-line comment.
= ~=	== /=	Equal and unequal. "=" in Haskell only means assignment operation, while "==" in Miranda is declaring a type synonym.
& \/	&& ||	Logic AND and OR. ".&." and ".|." are bitwise logic operators in Haskell.
!	!!	Indexing a list.
#	Length	Length of list.

Table 4-0-1 Major different operators in Haskell and Miranda

2 Conditionals

In Miranda, we use “if, otherwise” to achieve conditionals. Such as:

```
Testfunction a = 1, if a=1
               = 0, otherwise
```

In Haskell, there are usually three ways to achieve conditionals:

i. *If..then..else*.

This is the traditional way to achieve conditional (*If e1 then e2 else e3*). But it is obvious that when the program comes to multi-conditions or nested *if*, the “*if*” expression will become clumsy and hard to read

ii. Guards

Another way to achieve conditionals. Here is an example:

```
Testfunction a | a==1    = 1
               | a>1     =0
               | otherwise    = -1
```

The expression after “|” is the guard, and then the result for the function.

iii. Case expression

It is similar to “case” in other languages like Java. Here is an example:

```
take m ys          = case (m,ys) of
                   (0,_)      -> []
                   (_,[])     -> []
                   (n,x:xs)   -> x : take (n-1) xs
```

3 Type declaration

All Haskell types have to start with capital. In Miranda, they must not. In both Miranda and Haskell, we use “::” to declare a function’s type. In Miranda, we use “::=” to declare an algebraic type and “==” to declare type synonym. In Haskell, we use the keyword “data” or “newtype” to declare algebraic type and keyword “type” to declare type synonym.

4.1.2 Abstract type vs Type class

In both Miranda and Haskell, there is a way to define interfaces for types: abstract type in Miranda and type classes in Haskell. They look similar, both define several interfaces for types, but they are definitely not the same thing.

- A Miranda abstract type creates a new type with encapsulation (hidden representation and defined functions).
- In Haskell, typeclasses work for a set of types, providing common features for these types.

Typeclasses are the key features in Haskell, involving numeric operators, equality testing, etc. At some extent, it looks like the class in OO language—declare the class first, then create an instance. Type classes are also principles for overloading in Haskell, while abstract type in Miranda does not have this feature.

4.1.3 Type polymorphism and pattern matching.

Miranda and Haskell perform almost the same on pattern matching and type polymorphism. But Haskell has more features on this area:

- Wild card: when we match against a value we really care nothing about, we can use wild card. For example: head (x:_) = x.
- Case "foo x x = ….." : This case of pattern matching is legal in Miranda. It tells the case that the two parameters of function foo are equal. Whereas, it is not allowed in Haskell. If we want to achieve this, we have to add conditions or guards in the function. For example:

  foo a b | a == b =.....

  | otherwise =……
- Evaluation choices on pattern matching: Haskell provides "!" operator to force the evaluation and "~" operator to delay evaluation when matching patterns.
- Ad-hoc polymorphism: Compared to Miranda, Haskell has another kind of polymorphism –Ad-hoc polymorphism. This refers to when a value is able to adopt any one of several types because it, or a value it uses, has been given a separate definition for each of those types. For example, the function mod has the type Integral a => a -> a -> a. "a" is a polymorphic type bounded by (Integral a), which means "a" has to be an instance of typeclass Integral [24].

4.1.4 Library mechanism

Haskell's library mechanism is very different from Miranda's.

In Miranda, there are three common keywords serving the library mechanism: *%include* is to make included file's declaration and functions in scope to another file. *%insert* causes the contents of "file" to be substituted for the %insert direction during lexical analysis. *%export* is to specify "parts" to be exported to an including file.

In Haskell, source files represent as modules. Each module has a name definition, export list and import list:

module MODULENAME(EXPORT LIST) where

IMPORT LIST

Functions and declarations

The export list identifies the entities to be exported by a module declaration. The import list consists of import declarations. An import declaration brings an outside module into scope with exported entities, or specified import details. [25]

It should be noticed that there is no equivalent of Miranda's *"%export"* and *"%insert"* behaviors in Haskell. So it takes different approaches to organize source files between these two languages.

4.2 Codes restructure

4.2.1 Why does it need to be restructured?

Since Haskell has different library mechanism and module abstraction from Miranda, the original structure of the simulator is not suitable for continuing development. Thus, codes restructure is necessary.

4.2.2 Restructure implementation

Figure4-1 is the new structure of the simulator. Test cases is on the top-level. Comm is common module that record the basic types and functions shared across the simulator. There are four groups of modules that consist the body of the simulator. The "sim" group drives the simulation and provides statistic helper functions. The other three groups are the main components of the simulation. The module "Order" defines the functions for the order, and derives the order list (Sim_orderlist) and the limit order book (Sim_lob) which serve for the exchange agent. The module "Messages" defines functions for messages. The modules in agent group describe agents for the simulation.

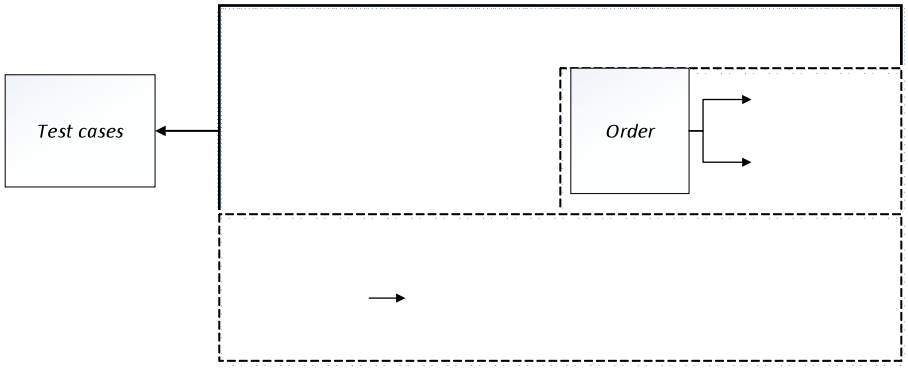


Figure 4-1 new module structure of the simulator

Isolated agents modules

In current simulator, there are six main implementations of agents: basic agents (including exchange agent, generic trader agent, spike agent and some agent for tests), Nicemime (exchange agent accurately mimics the behavior of the Chicago Mercantile Exchange's E-Mini market), Hft (high frequency trading market maker), LaggedHft: (Hft agent with lag), Fundamentals (fundamental trader), and Noiseagent.

These agents have the same type and share some parameters and helper functions. But they have their own features for certain simulation. In terms of future new agents deployed, it is necessary to divide the agent collection into two level: *Agents* as the base level which includes basic agents and shared basic parameters and functions; specified agents built on *Agents* designed for certain simulations, such as *Hft*.

Benefits:

- Lighter individual source files, easier to maintain
- Explicit reference on certain test. E.g. if we want to test with HFT and Nicemime, we just need to import these two agents in the test module.
- No dependency between specified agents. Except resources in Agents, there is no more sharing between specified agents. Thus the developer completely need not concern other agents' implementation when deploying a new agent.

Creating base module "*Comm*"

There are several related types, basic helper functions widely used in modules and type classes in the

simulator. To make it easier to maintain, collecting them into a new module (*Comm*) is a good way, similar to the header file in C.

Benefits:

- Managing helper functions to avoid code redundancy across modules.
- It is clearer to exhibit relations between types when we put them all in the same place. Also it is easier to maintain if there is modification on some type.
- *Comm* is the very base of the simulator. It easier for developers to absorb necessary information and resource in the simulator for their continuing work

Export list and Import management

In terms of safety, we should not let everything in a module can be seen by the outside. So export list is used for each module to protect private types and functions. On the other side, we should carefully import a module to another one. Import list should explicitly show which part of the module is imported to current one, and there is no unnecessary import. Import list should tell the programmer about all of the resources from outside used in current one.

Reserving type classes.

Though type class looks similar to abstract type in Miranda, it performs a very different role in Haskell program. In our Haskell simulator, there are five type classes. Each of them has only one instance. In our current situation, these five type classes logically indicate five important components in the simulator, but we didn't get any other benefit of using type class yet. But if the simulator get much more complicated in the future, we may benefit from type classes' features, such as ad-hoc polymorphism.

4.3 Porting result verification

Since the simulation is randomized, we cannot get the identical results from any two tests. To verify our new simulator in Haskell, it is fair that if simulator in both Miranda and Haskell run the same typical test and output similar results, we believe the simulator in Haskell is valid. More specifically, we investigate two important values in the output: Last Traded Price and Spread. If the Haskell

simulator is valid, the graphs of these two values should be similar to the result of simulation in Miranda.

I have run five trials of the tests for both versions, and it showed me the expected results. The ported simulator is valid. The detail of the test results is exhibited in appendix.

5 Simulator optimization in Haskell.

5.1 Profiling for Haskell

GHC (The Haskell compiler) provides a time and space profiling system [14]. With its cost center mechanism, GHC will record the cost of any given expressions at runtime and generate a call-tree of cost attribution (Cost centers are program annotations around expressions; all costs incurred by the annotated expression are assigned to the enclosing cost center). Thus, we can figure out which part of the program is the biggest consumer. An example profiling output is given in table5-2. Here is the explanation of the terminology that appears in the profiling result:

- Entries: The number of this particular point in the call tree was entered;
- Individual %time: The percentage of the total run time of the program spent at this point in the call tree;
- Individual %alloc: The percentage of the total memory allocations (excluding profiling overheads) of the program made by this call;
- Inherited %time: The percentage of the total run time of the program spent below this point in the call tree;
- Inherited %alloc: The percentage of the total memory allocations (excluding profiling overheads) of the program made by this call and all of its sub-calls.

5.2 Detecting hotspots

To get profiling result that reflect the real situation of the simulator, we need to run the simulation with typical practical experiments other than simple functional tests, so that we can precisely detect hotspots. We choose the typical test named testmeHetero which includes one Nicemime agent and

five Hft agents with 200 sim steps in default. The profiling test ran on a machine containing an Intel® Core™ i5-3337U processor running at 1.8 GHz with 4 cores. The OS was Window 7 64bit, and I chose to use Haskell platform 2013.2. .

5.2.1 First trial of profiling test.

Before we start repetitive profiling tests to get real detailed information of the simulator, it is necessary to get an initial snapshot of the situation. Here is the summary result of first trial of profiling test:

```
   total time  =        4.36 secs   (4357 ticks @ 1000 us, 1 processor)
   total alloc = 2,493,788,908 bytes  (excludes profiling overheads)

COST CENTRE     MODULE    %time %alloc

mymod           Agent      88.0   89.3
showmsg_t       Messages    2.2    2.2
showsimstate_t  Sim         1.4    3.0
showorder       Order       1.1    1.0
tracer          Sim         0.9    1.0
```

Figure 5-1 Summary result of first trial of profiling.

The summary gives us total cost and the rankings of costly functions (the total cost of functions). The result is surprising. A function named mymod consumes 88% time and 89.3% memory. It seems the simulator got stuck at this function, and we could barely get information about other functions because of mymod. We need to investigate mymod in more detail by exploring call-tree in profiling result and the implementation. Here is the part of the call-tree related to mymod:

```
                                                                    individual     inherited
COST CENTRE                                MODULE          no.     entries  %time %alloc   %time %alloc
nice_mime                                  Agent           605         200   0.0   0.0    90.4   90.7
 nice_mime.(...)                           Agent           622         200   0.0   0.0    88.9   89.6
  nice_mime_lob_appendorders               Agent           623        2433   0.0   0.0    88.9   89.6
   nice_mime_lob_appendorders.finallob     Agent           814        2233   0.0   0.0     0.0    0.0
   nice_mime_lob_appendorders.newlob       Agent           791        2233   0.0   0.0     0.0    0.0
   nice_mime_lob_appendorders.(...)        Agent           790        2233   0.0   0.0     0.0    0.0
   nice_mime_lob_appendorders.otheracks    Agent           789        2233   0.0   0.0     0.0    0.0
   nice_mime_lob_appendorders.fixedorder   Agent           743        2233   0.0   0.0    44.5   45.5
    safehd                                 Agent           748        2233   0.0   0.0     0.0    0.0
    getticksize                            Agent           746        2233   0.0   0.0     0.0    0.0
    fixtick                                Agent           744        2233   0.0   0.0    44.5   45.5
     fixtick.roundeddownprice              Agent           755          37   0.0   0.0     0.8    0.7
      mymod                                Agent           757      290875   0.8   0.7     0.8    0.7
      order_getprice                       Order           756          74   0.0   0.0     0.0    0.0
     order_setprice                        Order           754          37   0.0   0.0     0.0    0.0
     isOtype                               Agent           750          50   0.0   0.0     0.0    0.0
      isOtype.result                       Agent           751          50   0.0   0.0     0.0    0.0
       order_gettype                       Order           753          50   0.0   0.0     0.0    0.0
       isOtype.result.check                Agent           752          50   0.0   0.0     0.0    0.0
     order_getprice                        Order           747        2233   0.0   0.0     0.8    0.0
     mymod                                 Agent           745    17450934  43.7  44.8    43.7   44.8
    nice_mime_lob_appendorders.(...)       Agent           741        2233   0.3   0.1    44.4   44.0
    putintick                              Agent           837        2181   0.0   0.0    43.6   43.7
     isOtype                               Agent           842          16   0.0   0.0     0.0    0.0
      isOtype.result                       Agent           843          16   0.0   0.0     0.0    0.0
       order_gettype                       Order           845          16   0.0   0.0     0.0    0.0
       isOtype.result.check                Agent           844          16   0.0   0.0     0.0    0.0
     safehd                                Agent           841        5378   0.0   0.0     0.0    0.0
     order_getprice                        Order           840        4405   0.0   0.0     0.0    0.0
     getticksize                           Agent           839        2181   0.0   0.0     0.8    0.0
     mymod                                 Agent           838    17038299  43.5  43.7    43.5   43.7
    nice_mime_lob_appendorders.myupdatelob Agent           792        2170   0.0   0.8     0.3    0.1
```

Figure 5-2 Part of the call-tree of first trial of profiling

The call-tree shows there are only two places calling mymod, but mymod has much more massive entries than other functions, which means mymod is called much more frequently than other functions.

Here is the implementation of mymod:

```
mymod:: Double -> Double -> Double
mymod x 0 = error "mymod applied to zero"
mymod x y |(x >= 2*y)      = mymod (x-y) y
          |(y > x)         = x
          |otherwise       = x-y
```

This function is a version of mod for fractional numbers implemented in a recursive way. In our specific situation, the first argument of mymod is actually the price of orders and the second argument is a constant 0.25. Let us investigate a very common case "mymod 2000 0.25". Followed the recursive operations, mymod will be recursively called 8000 times. This explains why mymod has that huge amount of entries. Each call has an overhead cost, thus mymod wastes a lot of time and memory on its recursion. This is an inadequate implementation, or we can call it a "fault".

Mymod has to be redesigned before we continue further tests. The solution is simple – avoid the recursion. The improved mymod with new algorithm is exhibited below:

```
mymod x y   | y>x    = x
            | otherwise = x – ((fromIntegral(floor(x/y)))*y)
```

5.2.2 Second trial of profiling test.

Without the disturbance of recursive mymod, we run profiling tests again. Here is the result:

```
   total time  =        0.40 secs   (397 ticks @ 1000 us, 1 processor)
   total alloc = 209,757,956 bytes  (excludes profiling overheads)

COST CENTRE             MODULE   %time %alloc

showmsg_t               Messages  25.2   23.7
tracer                  Sim       14.6   14.0
showorder               Order     14.6   12.7
showsimstate_t          Sim       11.6   27.3
nice_mime_lob_gettrace  Agent      4.5    5.3
hftwrapper1.tracemsg    Agent      4.0    3.1
showsPrec               Messages   2.3    1.6
showsPrec               Order      1.8    0.6
showsPrec               Order      1.8    0.9
==                      Messages   1.0    0.0
showmsg_t.text          Messages   0.8    1.6
showsPrec               Order      0.8    1.5
```

Figure 5-3 Summary result of second trial of profiling

Compared to our previous analysis of hotspots in Miranda simulator, this profiling result is another

surprise. The suspected hotspots (in section 3.1.1) did not appear in the summary, and the top 4 consumers take over 66% time and 77.7% memory instead. Interestingly, top 4 functions are belong to a group that retrieves the simulation result and outputs to files. Showmsg_t, showorder and showsimstate_t are functions that convert target data type to strings for showing, and tracer is the top-level function in this group. More specifically, showsimstate_t operates a group of objects individually and put them together to output. Here is the schematic:

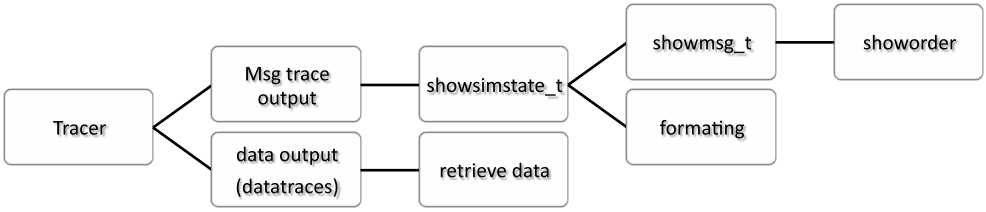


Figure 5-4 The schematic of Tracer

It seems like this group of functions are the hotspots with highest priority to be solved. But the information in this trial of profiling is far from enough for next optimization, we need a series of profiling tests which give us detailed break-down information of tracer and eliminates the impact of randomization to performance.

5.2.3 Full profiling results

To get a break-down information of tracer, I broke showsimstate_t into several sub-expressions so that the profiling tool can give us more detailed cost of it. Here is the altered showsimstate_t:

```
type Simstate_t = ([[Msg_t]], Int, [Msg_t], [[Msg_t]])
showsimstate_t :: Simstate_t -> [Char]
showsimstate_t (m,b,c,br)  = ", \n\nSystem Time: "++t++"\nMessages to sim: "++m2s++", \n\nHarness messages: " ++ h ++ "\n\nBroadcasts: "++bm++"\n\n\nEND OF STATE\n"
                                where
                                t = show b
                                bm = (concat (map (concat.(map showmsg_t)) (drop 1 br)))
                                h   = (concat (map showmsg_t c))
                                m2s = concat msg2Sim
                                msg2Sim = [f] ++ g
                                          where
                                          f = (concat.(map showmsg_t)) (head (drop 1 m))
                                          g = map (concat.(map showmsg_t)) (tail (drop 1 m))
```

We not only care about the performance on only one test situation, but also we need to know under

different test pressure what the result will be. Since we analyzed that there are two bottlenecks in the simulator, we should also pay attention to them, even though they did not appear as hotspots in previous profiling test. Thus, beside the top 4 functions and the two functions after top 4 which also consume considerable time and memory, I add sim_updatestate and nice_mime, which are the two bottlenecks (in section 3.1.1) and their cost need to be retrieved from call-tree, into our investigation. Since there are too many called functions under the two bottlenecks, we investigate the inherited cost of them, unlike the hotspots in summary.

I design two groups of tests:

1. testmeHetero, 400 steps, number of HFT agents from 5 to 10

Here is the result:

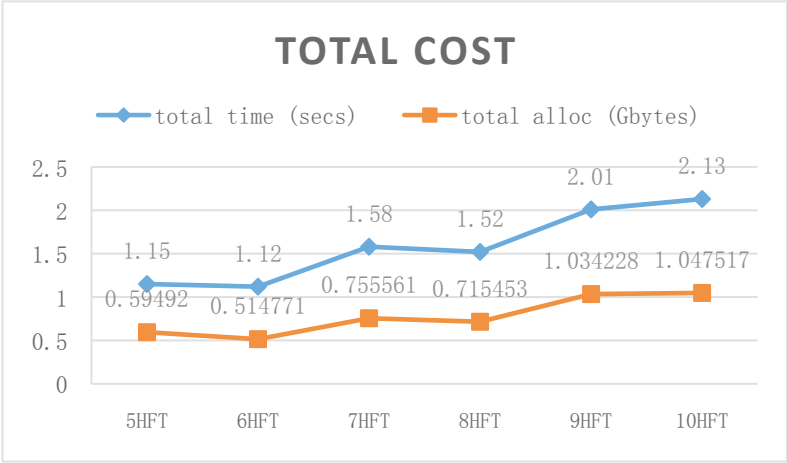
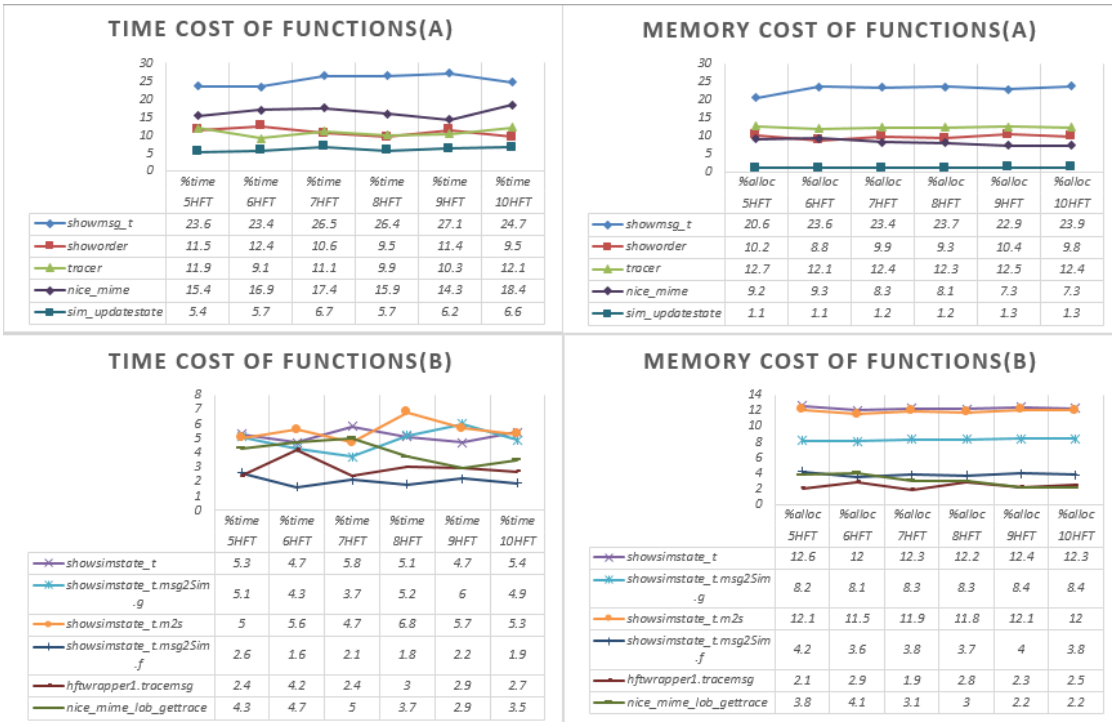


Figure 5-5, Graph of total cost of each test.



TIME COST OF FUNCTIONS(A)	%time 5HFT	%time 6HFT	%time 7HFT	%time 8HFT	%time 9HFT	%time 10HFT
showmsg_t	23.6	23.4	26.5	26.4	27.1	24.7
showorder	11.5	12.4	10.6	9.5	11.4	9.5
tracer	11.9	9.1	11.1	9.9	10.3	12.1
nice_mime	15.4	16.9	17.4	15.9	14.3	18.4
sim_updatestate	5.4	5.7	6.7	5.7	6.2	6.6

MEMORY COST OF FUNCTIONS(A)	%alloc 5HFT	%alloc 6HFT	%alloc 7HFT	%alloc 8HFT	%alloc 9HFT	%alloc 10HFT
showmsg_t	20.6	23.6	23.4	23.7	22.9	23.9
showorder	10.2	8.8	9.9	9.3	10.4	9.8
tracer	12.7	12.1	12.4	12.3	12.5	12.4
nice_mime	9.2	9.3	8.3	8.1	7.3	7.3
sim_updatestate	1.1	1.1	1.2	1.2	1.3	1.3

TIME COST OF FUNCTIONS(B)	%time 5HFT	%time 6HFT	%time 7HFT	%time 8HFT	%time 9HFT	%time 10HFT
showsimstate_t	5.3	4.7	5.8	5.1	4.7	5.4
showsimstate_t.msg2Sim.g	5.1	4.3	3.7	5.2	6	4.9
showsimstate_t.m2s	5	5.6	4.7	6.8	5.7	5.3
showsimstate_t.msg2Sim.f	2.6	1.6	2.1	1.8	2.2	1.9
hftwrapper1.tracemsg	2.4	4.2	2.4	3	2.9	2.7
nice_mime_lob_gettrace	4.3	4.7	5	3.7	2.9	3.5

MEMORY COST OF FUNCTIONS(B)	%alloc 5HFT	%alloc 6HFT	%alloc 7HFT	%alloc 8HFT	%alloc 9HFT	%alloc 10HFT
showsimstate_t	12.6	12	12.3	12.2	12.4	12.3
showsimstate_t.msg2Sim.g	8.2	8.1	8.3	8.3	8.4	8.4
showsimstate_t.m2s	12.1	11.5	11.9	11.8	12.1	12
showsimstate_t.msg2Sim.f	4.2	3.6	3.8	3.7	4	3.8
hftwrapper1.tracemsg	2.1	2.9	1.9	2.8	2.3	2.5
nice_mime_lob_gettrace	3.8	4.1	3.1	3	2.2	2.2

Figure 5-6 Graph of cost of functions

2. testmeHetero, 5 HFT agents, steps 200, 400, 800

Here is the result:

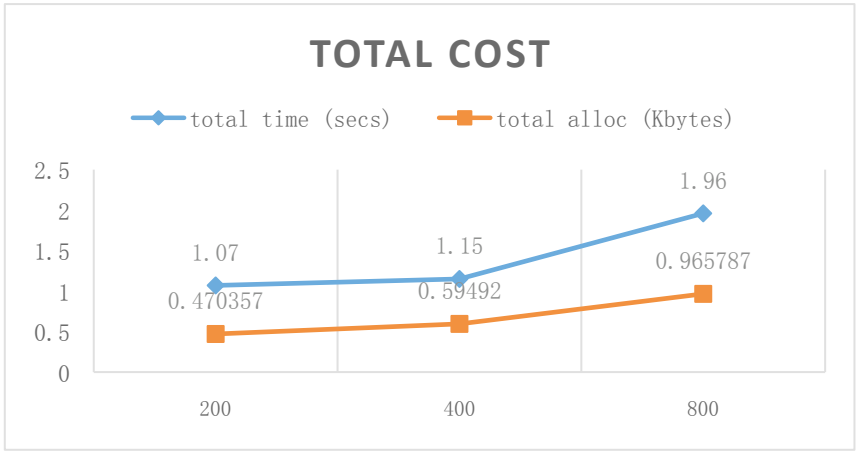


Figure 5-7 Graph of total cost of each test.

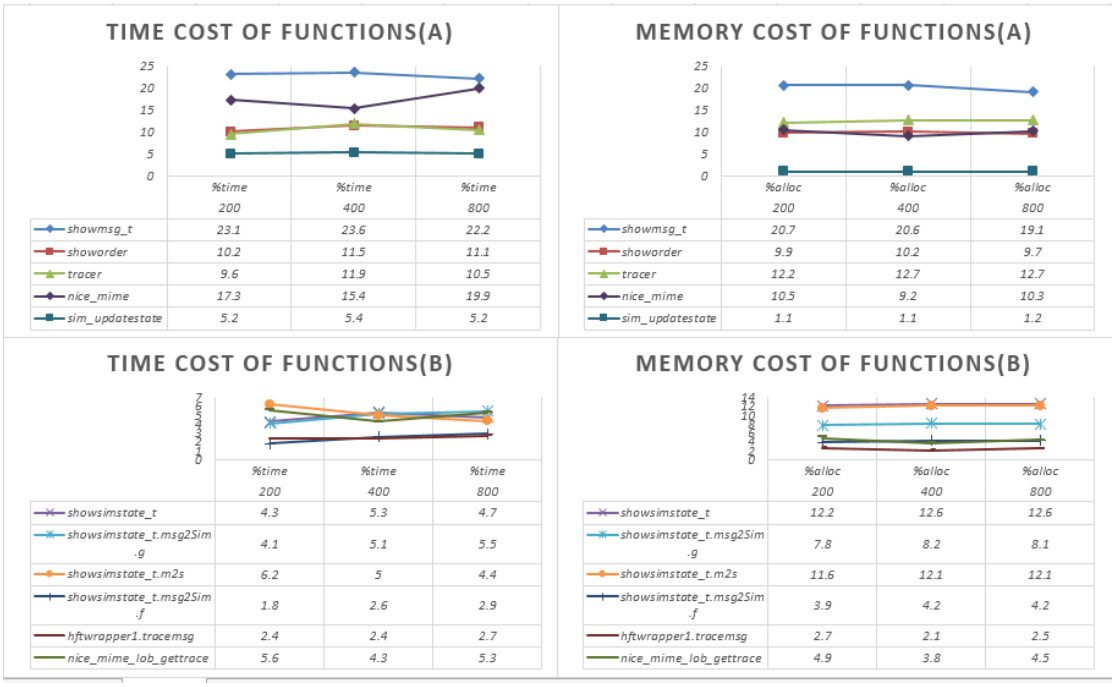


Figure 5-8 Graph of cost of functions

5.2.4 Hot spots analysis and conclusions.

Statistical analysis

We can see from figure5-5 and figure5-7 that total cost approximately linearly grows with both the number of agents and the length of the simulation (sim steps). But in figure5-5, if we increase an odd number of agents by one, such as 7 Hft to 8 Hft, the cost does not grow with it. It seems like even number of Hft agents and odd number of Hft agents are two different tests. We investigated the simulation output to get the truth behind this phenomenon. With similar approach to section4.3, we checked the graphs of Spread. It shows the Spread gets stable at zero faster in the even number of Hft agents cases than the odd number of Hft agents. When Spread get stable, there will be no trade in the market. Thus, the simulator would face a lighter workload (less calculation for order, less trade messages). Here are the graphs of 7Hft and 8Hft as an example:

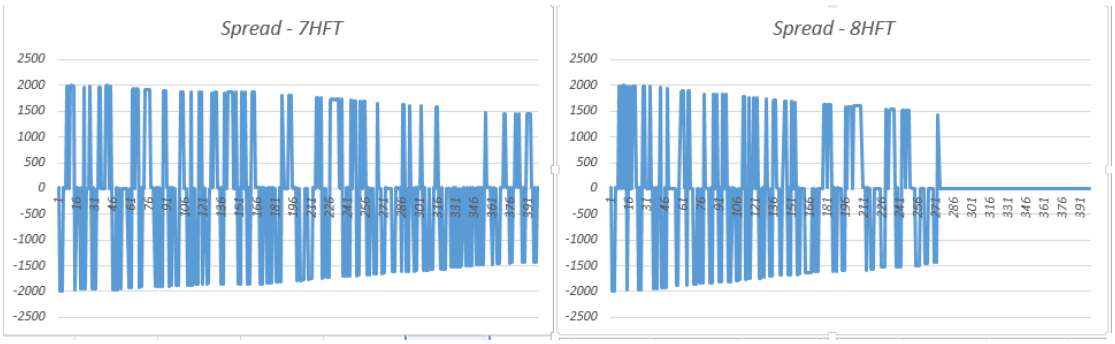


Figure 5-9 Spread graphs of tests with 7Hft and 8Hft

In contrast, the proportion of each function's contribution roughly stays at the same level (figure5-6 and figure5-8). Moreover, for our more interested tracer group, the ratio between those functions are relatively stable. E.g. the ratio of time cost of IO operation (indicated by tracer in the graph) and data retrieving (indicated by all other functions in this group) is roughly 1:5 (11.9 :53.1 in the 5HFT, 400 sim steps test). Since the phase of the simulation matters the workload, we are also interested in the workload contributions of agents in different phases. According to the design of the simulator, trader agent and exchange agent perform two absolutely different roles in the simulation, which trader agent sends order messages while exchange agent replies messages from trader agent and send trade messages if orders are executed. We investigate the workload in two groups: from exchange agent and from all traders. In our implementation for profiling, *showsimstate_t.msg2Sim.f* indicates the process for messages received by exchange agent, and *showsimstate_t.msg2Sim.g* indicates the process of messages received by all trader agents. The cost of these functions can reflect the volume of messages at some extent (actually messages sent by exchange and trader are different and cost different time and memory to convert them to strings). We can see from figure5-10 that the ratio is bounded around 2 to 3. If we look at the last three spots, the ratio decreases with the increase of the length of the simulation. In our test conditions, the second group of profiling tests has the same small amount of agents. The longer the simulation is, the more stable time will be. Thus, less trades causes less workload from the exchange agent, and the ratio decreases. It tells us: (i) the conversion of messages sent by exchange agents is more costly than ones sent by trader agents. Because the minimum ratio is around 2, it should be 1 if their costs are the same; (ii) the amount of messages sent by exchange agents can never exceed twice as much as the amount sent by trader agents, otherwise the maximum ratio should be over 3.

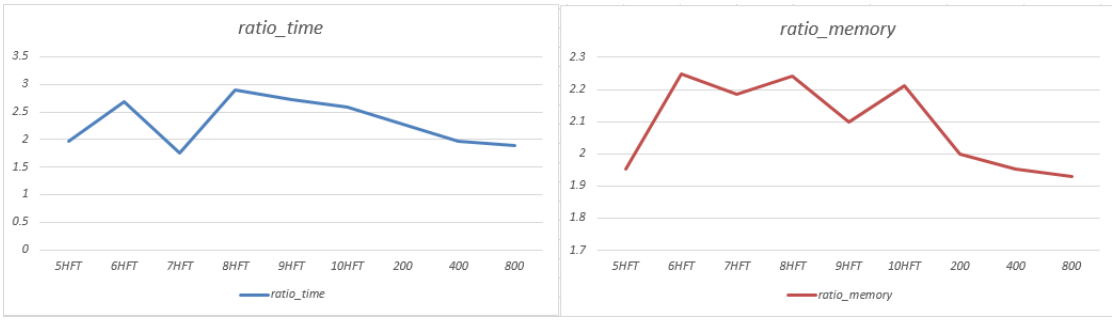


Figure 5-10 Cost ratio (*showsimstate_t.msg2Sim.g* : *showsimstate_t.msg2Sim.f)*

Tracer

As we discussed, tracer takes charge of the output of the simulator. It involves IO operation, data conversion and re-formation. Noticed that tracer will output all information during the simulation, the workload for this function is massive. Either IO operation or transferring data is costly. That is why tracer costs over half of the resources.

Broken-down tasks

Tracer operates two files individually. Thus the work can be broken down into two parts:

1. Simulation summary data output. There is a sub-function called *datatraces* doing this job. This function did not appear as hotspot in our profiling test, which means the cost of this path is negligible, compared to other functions.
2. Simulation messages trace output. It will print every messages communicating among agents in the simulation. There are three stages in this part (as the figure shows): (1) transfer the data in type of *msg* to string; (2) formatting strings for writing to file; (3) output to file. According to our profiling result, the most consuming part is transferring data (indicated by showmsg and showorder), and the cost of formatting and IO operation are very close (indicated by tracer and other function or sub-functions in this group)

5.2.5 Summary of hotspots

The Profiling result shows tracer and its subsequent functions consume at least 50% time and memory as the first group of hotspots. We should not ignore that nice_mime_lob_gettrace and hftwrapper1.tracemsg are also costly. Besides, our suspected bottlenecks (nice_mime and

sim_updatestate) do exist, but they are not that important problems compared to other hotspots.

5.2.6 Other hotspots

Beside the tracer group, we also detect out other less crucial hotspots:

1. nice_mime_lob_gettrace and hftwrapper1.tracemsg
   a) nice_mime_lob_gettrace converts statistics of limit order book to strings. It is similar to showsimstate_t, showmsg_t and showorder.
   b) hftwrapper1.tracemsg is converting data too. It retrieves Hft agent's current status and converts it to a data message.

   These two functions are mainly about data conversion. One possible solution is the same as the first solution for tracer (see section5.3), writing our own converting method. Since these two functions are not that urgent to be optimized, we can just leave them at the moment.
2. The two bottlenecks

   According profiling result, the two bottlenecks do exist, but they are not crucial too. The only way to solve it is to re-design this part. It is not worth to do it in this project for spending so too much effort and getting uncertain pay-back.

5.3 Solutions

Based on our analysis, there are two possible solutions to make tracer faster:

Data conversion is the biggest consumer in tracer, and it seems that the conversion by Haskell built-in method is not efficient enough. We could design our own converting functions to accelerate it. Considering the proportion of cost claimed by this part, a little improvement will bring a big increase in efficiency. Another possible solution is parallel computation. This is a straightforward solution that two cores would be faster than only one core to do the work. Fortunately, Haskell provides enough support for parallelism.

Compared these two solutions, I intend to deploy the parallelism. Here are my reasons:

1. The first solution theoretically looks good, but it is very difficult. I have to get very deep knowledge of Haskell and data conversion algorithm, if I want to deploy it. Besides we do not

know exactly how much the improvement will be until it is deployed, and there is no doubt that the effect is limited.

2. Parallel computation looks brilliant. Modern CPUs or other computing chips usually have more than one core. Programs have the ability of working with multiple cores is necessary. For our specific case, the work is not complex, so implementing parallel computation would not be too difficult with Haskell's support. The effect is also promising, one more core means there will be theoretical 100% improvement in ideal situation.

6 Parallelism implementation

6.1 Par Monad, a Haskell library for parallelism

The Par Monad is an open-source library developed in 2011 [26] to provide methods to achieve deterministic parallelism with explicit data dependency. Following the dataflow programming style, programmers can explicitly spark tasks across cores with the Par Monad. Another important benefit of Par Monad is that it allows programming tuning the scheduler for parallel task. We can gain more control through this to make our paralleled program more efficient. But we should also be noticed that this library has not been well optimized on the GHC side yet. "We examined the code generated by GHC for the Par monad version and observed several opportunities for optimization that are not yet being performed by GHC's optimizer". [26]

For our case, tracer handles several individual tasks and has an explicit data dependency. The Par Monad is perfect for us to achieve different levels of parallelism on tracer or even the whole simulator.

6.2 Granularity and parallel utilization

Theoretically, a job done by two cores should be twice faster than just by one core in ideal situation. That is **100% utilization** which two cores (or maybe more) always keep working and never need to wait each other. In fact, it is almost impossible to be achieved due to (i) uneven distributed tasks for each core; (ii) asynchronous garbage collection; (iii) uncertainty caused by IO operation; etc. Utilization is a standard to measure the effect of parallelism; (iv) "setup" overhead for tasks.

To achieve better utilization, we need to put more thought on distribute workload to task, which is task's **granularity**. If the granularity is too small, the overhead of forking task becomes significant. While if the granularity is too large, one core may be waiting to start its next work until the other core finished current work. That is underutilization. Thus, the granularity should be "just right".

6.3 Paralleled tracer

The major workload for tracer comes from the function showsimstate_t. We could try to parallelize showsimstate_t before we move to upper-level parallelism for tracer.

6.3.1 Straightforward parallelism on showsimstate_t

Let us recap the implementation of showsimstate_t. It operates a tuple *(m,b,c.br)*. *m* is the list of lists of messages communicating among agents. *b* is simulation time. *c* is the list of debug messages from "simstep". *br* is the list of lists of broadcast messages. They are independent. We can fork four tasks – one for each element in the tuple. The data flow is showed below:

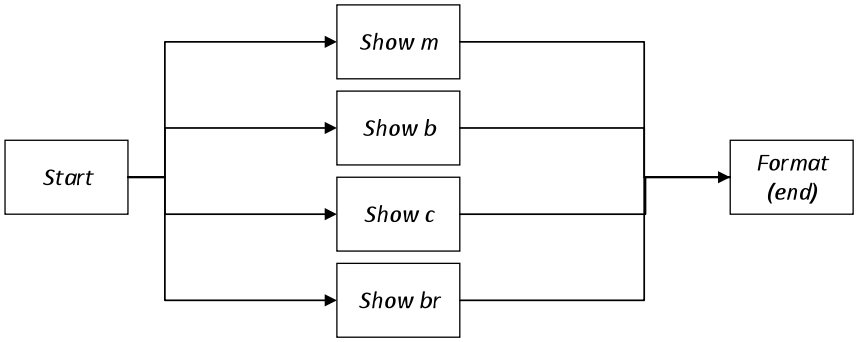


Figure 6-1 Data flow for straightforward parallelism on showsimstate_t

6.3.2 Static partitioning on showsimstate_t

Investigated more carefully on *(m,b,c,br)*, I found out that the workload of *m* is much heavier than others. Because messages from agents consist the core of the simulation. If we follow the solution in section 6.3.1, the utilization is obvious very low. Thus, we adjust the granularity. Here is the new dataflow:

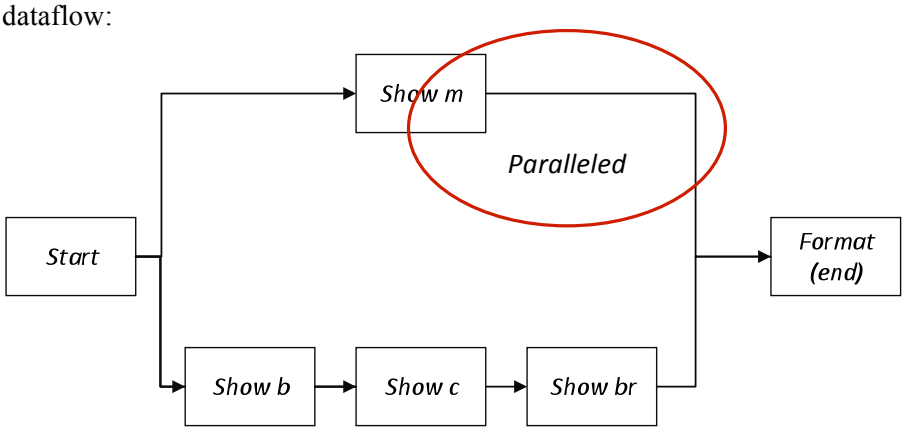


Figure 6-2 Data flow for static partitioning solution for parallelizing tracer.

Do these two paths have roughly even workload? The answer is no. Looking at profiling results, we can find out that, the workload of *m* is much heavier than other three combined together. We need make "show m" paralleled. A simple solution is static partitioning the list – *m*. There are two strategies for partitioning: (i) split *m* in half, fork a task for each half; (ii) spark tasks for every element in the list. Obviously, the granularity is small in the first strategy and may be too large in the second strategy.

6.3.3 Dynamic partitioning on showsimstate_t

Static partitioning does not distribute the workload properly. The length of m equals to the number of agents, and the amount of messages sent by each agent depends on which phase the simulation is in. More specifically, according to the design of simulator, exchange agents will reply to every message sent by trader agents and send trade messages if there are executed orders. Thus, the ratio of messages sent by exchange agents and messages sent by all trader agents is between 1:1 (no executed order) and 2:1 (every order issued by trader agents is executed). The profiling results in section5 actually proved this point (section 5.2.4, Statistical analysis).

Having known these information, we could dynamically partition the workload of *m* to get more appropriate granularity. However, this solution is not implemented at the moment.

6.3.4 Pipeline mechanism on tracer

Obviously, only having paralleled showsimstate_t is far from enough to get a good utilization. We can deploy parallelism from the top-level of tracer. The data dependency in tracer is explicit. Tracer will not output data until the data is completely evaluated. We can handle the two output paths in parallel, but that is obviously not enough. According to the profiling result, the workload ratio of tracer's IO operation and data retrieving is roughly 1:5 (indicated by tracer and showsimstate_t and its called functions, section5.2.4). We can introduce pipeline parallelism. For example, we buffer the data at time t and output it at time t+1, so that the data retrieving and IO operation can be paralleled, as the figure shows:

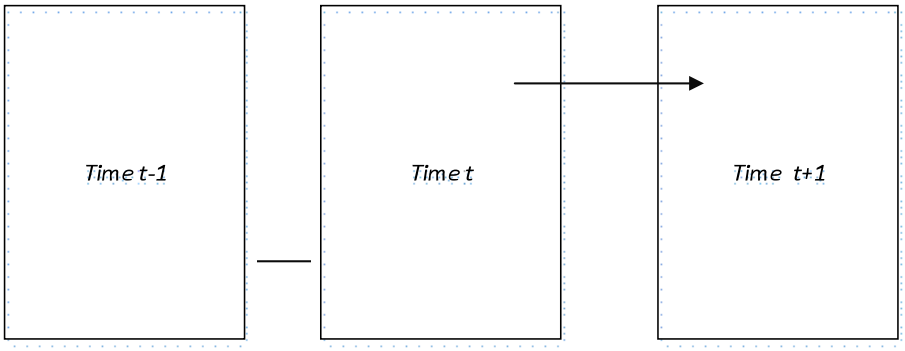


Figure 6-3 pipeline for tracer

Actually, we can change the buffer strategy, such as buffering data of two time steps, to find an appropriate granularity. This solution requires a re-design of the tracer. Hence, I did not implement it in this project.

6.3.5 Summary of paralleled tracer

We introduced four possible solutions for parallelizing tracer.

Solutions	Expected performance	Difficulty
Straightforward parallelizing showsimstate_t (section 6.3.1)	It might not distribute workload well. Bad granularity and low utilization.	Easiest to implement
Static partitioning on showsimstate_t (section 6.3.2)	We gave two strategies for this solution. Either of them may perform well on some tests. But it might not suitable for other tests.	Easy to implement. We partition the workload following a static strategy, then fork tasks.
Dynamic partitioning on showsimstate_t (section 6.3.3)	A self-adaptive partitioning algorithm works for this solution. It would bring suitable granularity for any tests.	Need to design the algorithm and observing mechanism for the simulation to get input for the algorithm. It is much more complex than the previous solutions.

Pipeline mechanism on tracer (section 6.3.5)	Higher-level parallelism. Easy to control the granularity. It is the best solution for tracer.	Need a re-design of the tracer.

Table 6-1 Comparison of four parallelism solutions for tracer

Due to the time limit of the project, we only implemented first two solutions (including two strategies in the second solution). Other solutions have to be left to the future development.

6.4 Profiling the paralleled simulator

6.4.1 ThreadScope – a tool for profiling paralleled Haskell program.

We care more about the scheduling information of multi-threaded programs than single-threaded programs. The GHC profiling tool does not provide information on that area. Fortunately, another tool is provided specifically for profiling paralleled Haskell program. ThreadScope visualizes the event log dumped from Haskell runtime to give us a view of tasks scheduling across processors, so that we can know the utilization level, garbage collection status and load balancing issues.

6.4.2 Profiling results for paralleled simulator

Even though ThreadScope is not for profiling single-threaded program, it is necessary to profile our single-threaded simulator with ThreadScope as a comparison. I chose the typical test testmeHetero with 200 simulation steps and 5 HFT agents for profiling. In order to mitigate the oscillation brought by randomization of the simulation and instability of the test machine (IO, scheduling), I ran ten times for each test, and picked up the most common trial to show below.

- Result for single-threaded similar

INIT time 0.00s (0.00s elapsed)
MUT time 0.20s (0.21s elapsed)
GC time 0.05s (0.03s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 0.27s (0.24s elapsed)

%GC time 17.6% (10.9% elapsed)
Alloc rate 919,423,889 bytes per MUT second
Productivity 82.4% of total user, 90.7% of total elapsed

The first part gives us the CPU time and wall clock time elapsed broken down into five parts: *INIT* is the runtime system initialization. *MUT* is the mutator time, i.e. the time spent actually running the code. *GC* is the time spent doing garbage collection. *EXIT* is the runtime system shut down time. *Total* is, of course, the total. [27]

In the second part, *%GC* tells what percentage GC is of Total. *Alloc rate* tells you the "bytes allocated in the heap" divided by the MUT CPU time. *Productivity* tells what percentage of the Total CPU and wall clock elapsed times are spent in the mutator (MUT). [27]

- Result for the solution straightforward parallelism on showsimstate_t (section6.3.1):

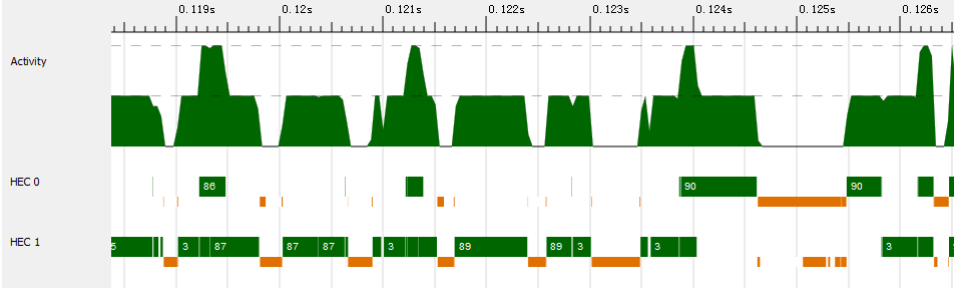
  The test ran with two cores:

  Parallel GC work balance: 16.80% (serial 0%, perfect 100%)
  INIT time 0.00s (0.00s elapsed)
  MUT time 0.36s (0.33s elapsed)
  GC time 0.25s (0.17s elapsed)
  EXIT time 0.00s (0.00s elapsed)
  Total time 0.61s (0.51s elapsed)

  Alloc rate 726,365,477 bytes per MUT second
  Productivity 59.0% of total user, 70.9% of total elapsed

  Instead of *%GC,* the result for paralleled version gives us *Parallel GC work balance* indicating the paralleled GC work balance. 0% means totally serial GC, while 100% means all GC works in parallel all the time.

  We see that this solution is much slower than the single-threaded version. GC cost is much higher, and the productivity decreased from 90.7% to 70.9%. Let us check the thread scheduling information:



Figure

6-4 ThreadScope result for straight forward parallelism with two cores

The *HEC* row indicates the thread scheduling of a core. We can see that there are two cores working for this program. In *HEC* row, the thick bars represent threads with thread numbers on them, while the thin bars represent garbage collection. The Activity row indicates utilization.The dash lines in *Activity* row to illustrate two cores utilization level. The utilization is awful. There was only one core working for most of the time. We can clearly find out that the workload distributed extremely uneven. This parallel implementation is bad.

- Result for solution of Static partitioning on showsimstate_t (section6.3.2):

  I proposed two strategies on static partitioning. Here is the test result for both strategies:

  1) Split m in half.

  - Test with two cores, five Hft agent:

```
Parallel GC work balance: 21.85% (serial 0%, perfect 100%)
INIT        time    0.00s  (  0.00s elapsed)
MUT         time    0.37s  (  0.22s elapsed)
GC          time    0.03s  (  0.06s elapsed)
EXIT        time    0.00s  (  0.00s elapsed)
Total       time    0.41s  (  0.28s elapsed)

Alloc rate    645,561,620 bytes per MUT second
Productivity  92.3% of total user, 133.7% of total elapsed
```

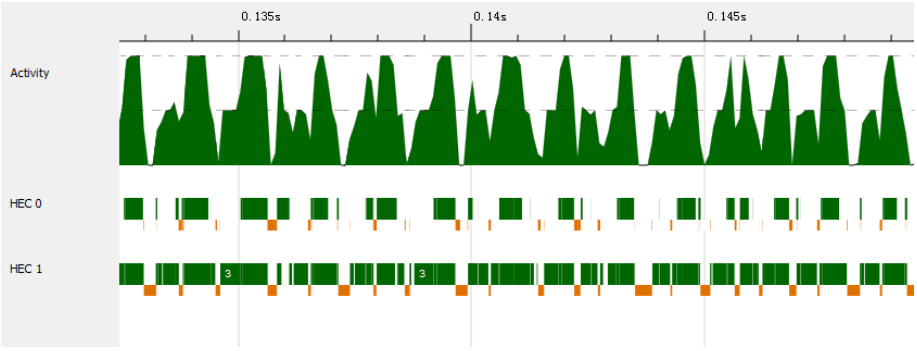


Figure 6-5 ThreadScope result for “split in half” with two cores, five Hft agents

Compared to the straightforward solution, the performance is significantly improved. Utilization is much better and the cost of GC plunged.

- Test with three cores, five Hft agent:

  We may be wondering what if we add one more core, what will happen. Here is the result:

Parallel GC work balance: 9.29% (serial 0%, perfect 100%)
INIT time 0.00s (0.00s elapsed)
MUT time 0.23s (0.19s elapsed)
GC time 0.12s (0.05s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 0.37s (0.24s elapsed)

Alloc rate 807,077,664 bytes per MUT second
Productivity 66.7% of total user, 104.3% of total elapsed

The performance stays at the same level. It is reasonable. Because we only spark two tasks for paralleled showsimstate_t. Theoretically, we do not need the third core for this strategy. The graph proved this point:

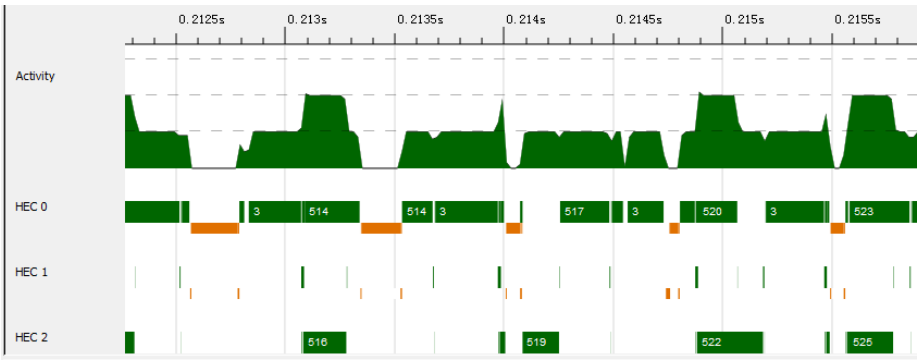


Figure 6-6 ThreadScope result for "split in half" with three cores, five Hft agents

In the graph, there was nearly no period that has the utilization level occupying all three rows in *Activity*.

2) Spark tasks for every element in the list.

- Test with two cores, five Hft agents:

Parallel GC work balance: 26.39% (serial 0%, perfect 100%)
INIT time 0.00s (0.00s elapsed)
MUT time 0.22s (0.21s elapsed)
GC time 0.06s (0.04s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 0.28s (0.26s elapsed)

Alloc rate 984,298,324 bytes per MUT second
Productivity 77.8% of total user, 85.5% of total elapsed

The performance (Total 0.26s elapsed) is as good as the first strategy (Total 0.28s elapsed)

in this particular test condition. But we should notice that *Parallel GC work balance* is better than previous strategy (26.39% to 21.85%). This strategy has smaller granularity than the first one. It should performance well on parallel GC.

- Test with three cores, five Hft agent:

```
Parallel GC work balance: 15.78% (serial 0%, perfect 100%)
INIT    time    0.00s  (  0.00s elapsed)
MUT     time    0.50s  (  0.30s elapsed)
GC      time    0.08s  (  0.06s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    0.58s  (  0.36s elapsed)

Alloc rate    546,873,241 bytes per MUT second
Productivity  86.5% of total user, 137.1% of total elapsed
```

In the three cores test, the performance went bad because of the big increase on MUT time. We mentioned that this strategy has more tasks than the first one. When we have an extra core, there will be extra cost on sparking tasks on that core and communication between cores.

6.5 Summary of parallelism implementation

Here is the summary based on the profiling result:

1. Parallelism with Par Monad has a considerable overhead. We can see from profiling results that our current parallelism implementation was not faster than the single-threaded version on the *testmeHetero* test. The cost of garbage collection increased enormously, which is mainly caused by sparking tasks. Our partially implemented parallelism cannot cover this overhead. The utilization level is very low.
2. We did improve the parallel performance by more reasonably distributing tasks.
3. The granularity of the implemented solution is too small. We can find out that there are too many tasks sparked. According to the implementation, showsimstate_t will spark parallel tasks at each simulation time step. For example, in the first solution of static partitioning, showsimstate_t sparked 400 tasks during the *testmeHetero* test.
4. To get an acceptable efficiency, we need to design the parallel implementation from top-level, so

that we can have enough “room” to adjust the granularity.

5. The fully implemented parallelism should be able to handle different test conditions and different amount of available cores.

7 Final result verification

We benchmarked on a Mac laptop containing an Intel® Core™ i5 processor which has two cores and run at 1.6GHz. The OS was Mac OS X Lion 10.7.5. We used a fixed heap size (100MB) for simulators in both Miranda and Haskell. Since the non-paralleled version of simulator in Haskell is stable and fastest at the moment, we used this version in benchmark.

7.1 Comparison tests for Miranda and Haskell versions.

To confirm the improvement brought by this project, we both simulators with the same experiment under the same environment. Since Miranda does not have profiling tools, we used the *time* command in UNIX to get the time cost for both simulators. We still use the test *testmeHetero* with 5 Hft agents and 200 sim steps. We ran 15 trials for both simulators. Here is the result (detailed results is in appendix2.2):

- Simulator in Miranda

The average time cost (real) is 75.249s.

- Simulator in Haskell

The average time cost (real) is 0.309s

We can see that the simulator in Haskell is about 243 (75.249/0.309) times faster than the simulator in Miranda. Because there are variations on results of both versions of simulator, I also ran a T-test to what extent these two distributions overlap. The "p value" of the result is $3.295*10^{-8}$. It is a very small value, which means it is almost impossible that the two distributions overlap.

7.2 The pressure test for Haskell version

We enormously increased the size of testmeHetero to get view of current capability of the simulator in Haskell. We increased the number of agents to 21, and sim steps to 800. Here is the result (from Haskell runtime):

INIT time 0.00s (0.00s elapsed)
MUT time 3.30s (3.44s elapsed)

GC time 1.65s (1.68s elapsed)
EXIT time 0.00s (0.00s elapsed)
Total time 4.95s (5.12s elapsed)
%GC time 33.3% (32.8% elapsed)
Productivity 66.7% of total user, 64.5% of total elapsed

It cost only 5.12 seconds to finish this much bigger test.

7.3 Summary for Haskell simulator

According to the verification result, the new simulator is over 200 times faster than the original simulator in Miranda. It can easily process much bigger simulations than what we had run before. Moreover, simulator's well-designed code structure with explicit module relations provides big convenience for future extension and development. Although current parallel implementation for the simulator is not faster than the non-paralleled version, it is still a meaningful exploration to be a good base of future parallel implementation for the whole simulator.

8 Summary and conclusion

8.1 Summary and recap contribution

Through this project, I experienced the course of entering an unknown world and then peeling the onion to figure out what exactly the problem is. Following extensive reading, I finally decided to use Haskell as the programming language for this project, which balanced the competing tension between the requirement and project time limitation. It is delightful that after I started writing Haskell code, I found out that Haskell has a solid support for real parallel programming. This brings a much brighter future of the simulator that not only the computation can be paralleled, also the whole simulation can be performed in an entirely parallel way not just pseudo BSP model.

Instead of a detailed analysis on the simulator in Miranda, I chose analysis on the ported simulator directly to find out the hotspots for the next optimization. With powerful profiling tools and extensive profiling tests, I detected out all hotspots in the ported simulator (section5.2.5) and proposed possible solutions for them (section 5.3).

Instead of struggling on optimizing certain functions, I chose a systematic way to improve performance of the simulator. Due to the time limit of the project, the parallel implementation still needs massive efforts on it. But on the bright side, the solution of parallelism is clear, and there is a good foundation built in the simulator during this project.

8.1.1 Review objectives

To review this project, all objectives set at the beginning have been met successfully:

1 O1: Haskell is the chosen language. It has almost all features of Miranda and much more powerful resources including new invention on functional programming, huge library collections and parallel programming support.

2 O2: A new simulator in Haskell is built with better code structure, and validated by validation tests.

3 O3: The optimized simulator is much faster (over 200 times) than the version in Miranda.

4 O4: Initial parallelism is deployed on the simulator. Even though it did not reach our expectation, the solution was figured out in this project.

8.1.2 Recap of contributions

- I have ported the simulator from Miranda to Haskell, with user manual and guidance for future development.
- The optimized simulator is much faster that the old one. We can run bigger and more complex simulation on it.
- Section 4.1 provides basic guidance of how to port Miranda program to Haskell.
- Section 5 provides a guidance of how to profile sequential Haskell programs and section 6.4 provides a basic guidance of how to profile parallel Haskell programs.
- Massive profiling tests and analysis afterwards help to get a deeper understand of how the simulation runs.
- I have introduced parallelism on the simulator. Even though the performance is not good enough, it illustrated a way of improving performance and a possible direction for the simulator's evolution.

8.2 Discussion on further optimization and development

It seems data conversion is costly in Haskell. According to the profiling result, most top hotspots are about data conversion, more specifically, related to the Haskell built-in function "show" which is used to convert data to string. Of course, there is massive data conversion workload there. But it is not reasonable that this work is much more costly than the bottlenecks in the simulator. We should optimize it in the future.

It seems the improvement of paralleled showsimstate_t cannot cover the overhead brought by parallel implementation. With static partitioning strategy, the improvement is not apparent. Although we can use a dynamic partitioning strategy to get higher parallel utilization, it is still doubtful whether it could cover the overhead. We should deploy parallelism from higher-level.

8.3 Conclusion

The non-paralleled version simulator can fulfil current needs of experiments. It is at least 200 times faster than the simulator in Miranda, and it can finished a simulation with 1 exchange agents and 21 Hft agents in 10 seconds (depending on the speed of the machine). This project pointed out the direction of future development of the simulator. Profiling tests detected out all hotspots, and then I analyzed them to explain why they are costly. Parallelism is a systematic solution to improve the performance. We are still trying to cover the overhead brought by the implementation. Profiling result indicates that we should consider the parallel implementation for the whole program, not just a part of it. Otherwise, we will never get an acceptable level of utilization and performance.

References

[1] E. Court, "The instability of market-making algorithm – an agent based simulation in Miranda," *MEng dissertation, Dept. Computer Science, UCL,* 2013.

[2] A. Getchell, "Agent-based Modeling," *University of California, Davis: Department Physics,* 2008.

[3] CFTC-SEC, "Findings regarding the market events of May 6, 2010.," in *Tech. Rep. 202, U.S. Commodity Futures Trading Commission and the U.S. Securities & Exchange Commission.*, 2010.

[4] J. Hughes, "Why Functional Programming Matters.," *In D. Turner, editor, Research Topics in Functional Programming, Addison-Wesley, ,* 1990.

[5] D.A.Turner, "SASL Language Manual," *St Andrews University Technical Report,* December 1976.

[6] H.Richards, "An overview of ARC SASL," *SIGPLAN Notices October,* 1984.

[7] D.A.Turner, "Recursion equations as a programming language," *Functional Programming and Its Applications, ed Darlington et al., CUP,* 1982.

[8] D. A. Turner, "An Overview of Miranda," *SIGPLAN Notices 21(12):158-166,* December 1986.

[9] R.Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences, vol 17,,* 1978.

[10] P. Hudak, J. Hughes, S. Peyton Jones and P. Wadler, "A History of Haskell: Being Lazy with Class," *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III): 12–1–55. doi:10.1145/1238844.1238856. ISBN 978-1-59593-766-7.,* 2007.

[11] "Haskell 2010 language report," [Online]. Available: http://www.haskell.org/onlinereport/haskell2010/.

[12] S. P. Jones, M. Jones and E. Meijer, "Type classes:an exploration of the design space," *In Launchbury, J., editor,Haskell workshop, Amsterdam.,* 1997.

[13] "Foreign Function Interface - Haskell 2010 report," [Online]. Available: http://www.haskell.org/onlinereport/haskell2010/haskellch8.html#x15-1490008.

[14] "GHC user's guide," [Online]. Available: http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html.

[15] "Cabal," [Online]. Available: http://www.haskell.org/cabal/.

[16] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM, Volume 33 Issue 8,* August 1990.

[17] "F#-Wiki," [Online]. Available: http://en.wikipedia.org/wiki/F_Sharp_(programming_language).

[18] "F# - The F# software foundation," [Online]. Available: http://fsharp.org/.

[19] "What is Scala," [Online]. Available: http://www.scala-lang.org/what-is-scala.html.

[20] "What is Ocaml," [Online]. Available: http://ocaml.org/learn/description.html.

[21] "Green Card home page," [Online]. Available: https://github.com/sof/greencard.

[22] "C->Haskell home page," [Online]. Available: https://github.com/haskell/c2hs.

[23] "The OCaml system release 4.01," [Online]. Available: http://caml.inria.fr/pub/docs/manual-ocaml-4.01/index.html.

[24] "Polymorphism - Haskellwiki," [Online]. Available: http://www.haskell.org/haskellwiki/Polymorphism.

[25] "Modules - Haskell," [Online]. Available: http://www.haskell.org/onlinereport/haskell2010/haskellch5.html#x11-990005.1.

[26] R. N. S. P. J. Simon Marlow, "A Monad for Deterministic Parallelism".*Haskell Symposium 2011.*

[27] "Running a compiled Program - Haskell," [Online]. Available: http://www.haskell.org/ghc/docs/latest/html/users_guide/runtime-control.html#setting-rts-options.

[28] "Profiling - GHC user guide," [Online]. Available: http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html.

[29] C. M. &. E. P. Chris Clack, Programming with Miranda, Prentice Hall, 1995.

Appendices

1 User and system manual

1.1 Overview of simulator

There are 17 modules in current simulator that can be divided into three categories:

1. Essential components.

   Comm, Baserands, Sim, Order, Sim_orderlist, Sim_lob, Messages and Stats belong to this category. These modules describe how the simulation runs, by defining data types, communication mechanism and operations.

2. Agent family.

   Agents, Nicemime, Hft, LaggedHft, Fundamentals and Noiseagents belong to this category. These modules relate to intended experiments. We model specific agent, and run it with the simulation. Agent family is open, developers can add their own agents.

3. Test cases

   TestHarness and Main belong to this category. These modules set up and run the simulation, totally depending on the needs of simulation. TestHarness is the collection of test cases, while Main indicates which test will be running in the program.

1.2 Getting start with simulator

First of all, download corresponding Haskell Platform for your computer (Windows? Mac? Or Linux) from the website *http://www.haskell.org/platform/*. Follow the instructions on the website to install Haskell Platform on your computer. The Haskell Platform packed Glasgow Haskell Compiler (GHC), GHC interpreter, GHC runtime, a collection of libraries and several developer tools, such as cabal and

Haddock. More details can be found in *http://www.haskell.org/platform/contents.html*. It is not recommended to manually update GHC version other than the version provided in Haskell platform. Everything in Haskell platform is well tested and stable.

GHC has three main components:

- ghc is an optimizing compiler that generates fast native code.
- ghci is an interactive interpreter and debugger.
- runghc a program for running Haskell programs as scripts, without needing to compile them first.

In ghci, we load our Haskell modules, then we can call any declared function from loaded modules. While with ghc, we may set some flags for compiling. Once compiler succeeded, we can run the executable directly. More details of GHC can be found in GHC user guide:http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html.

1.2.1 How to run a test.

The main function is and has to be the top-level function. Add the intended test function to the main function, then we start compiling. We use O2 optimization and enable Run Time System (RTS) options. The compiling command is:

ghc –O2 –rtsopts Main.hs.

for parallel version: ghc –O2 –threaded –rtsopts Main.hs

Then we can execute the compiled executable, which named Main.

1.2.2 How to profile the simulator.

To profile single-threaded version, we use GHC built-in tools. There is no alternation in code, but we need add some flags for compilation. Here is the compiling command:

ghc –O2 –rtsopts –prof –fprof–auto Main.hs

–prof is to enable profiling. *–fprof–auto* is to let compiler automatically add cost centers to all the functions in the program. Then, we run the program with the command

./Main +RTS –p

We tell run time system generate profiling result. The result file is named Main.prof.

More details about profiling with GHC can be found on GHC user guide chapter5. Profiling: (http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html#cost-centres)

It does not make much sense to use cost-center profiling multi-threaded programs. To profile parallel version, we ask run time system give us run time cost information and event log. Here is the compiling command:

ghc –O2 –rtsopts –threaded –eventlog Main.hs

To run the program, we enter the command

./Main -+RTS –s –ls

-s is to print out run time cost information. –ls is to generate event log named Main.eventlog.

The raw event log is just a text records of events during program execution. It is hard to digest to get comprehensive view. We use ThreadScope to visualize it. Run threadscope and load Main.evenlog manually. It will generates the graph for us.

1.3 Par-monad instructions

The Par-monad library may not be included in the Haskell Platform library collection. We need install it manually if we want to run parallel version. Ensuring the internet works well, at cmd window (Terminal in Mac), type following command to install the library:

cabal update

cabal install monad-par

It use the tool *cabal* to retrieve the library from hackage (a pool of thousands Haskell libraries, http://hackage.haskell.org/), and install it into Haskell Platform. The detailed library interface description can be found in http://hackage.haskell.org/package/monad-par-0.3.4.6/docs/Control-Monad-Par.html (the latest version 0.3.4.6).

To use the Par-monad, we need to import it first. Add the following statement at the start of the module which want to use it.

import Control.Monad.Par

When we use the Par-monad, there are two basic issues we should be aware of. The first issue is

parallel computations are pure and forked explicitly. With the functions *fork* and *spawn (actually there are more choices, check details in library description),* we can explicitly distribute the work in parallel tasks. The second issue is the data communicated between parallel tasks should be able to be fully evaluated. Parallel tasks communicate through IVars(a special data type). One task use function *put* to write data in an IVar, and other tasks can read the data from this IVar using the function *get.* To be noticed, the function *put* is fully strict. Thus we should be careful of the impact caused by this strict evaluation when we deploy the parallelism.

2 Tests results

2.1 Porting result verification

Once finished porting code from Miranda to Haskell, we need to verify that the ported simulator works as what we expect. I ran 5 trials for both Miranda and Haskell simulator under the same test condition. I investigated two typical values (Spread and Last Traded Price) on the trend of their graphs to find out whether the Haskell simulator output the expected result as the Miranda simulator did. Here is the result:

- Graphs of the Miranda simulator's output.

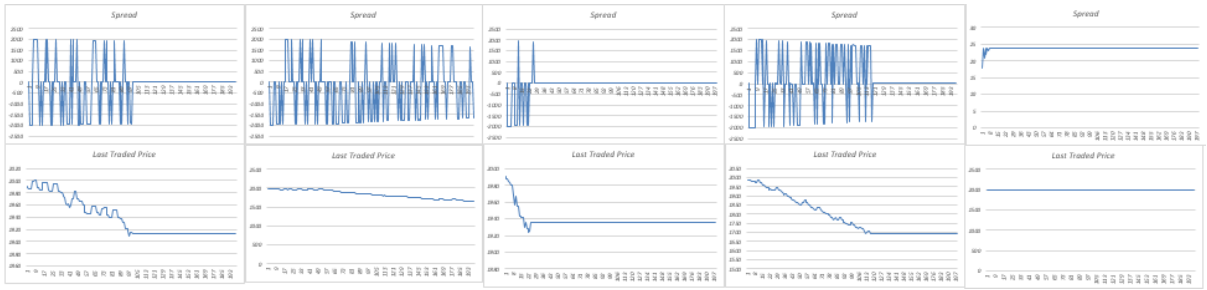


Figure A2-1, The Miranda simulator's output. The first row contains five graphs of Spread, and the second row contain five graphs of Last Traded Price (LTP).

- Graphs of the Haskell simulator's output.

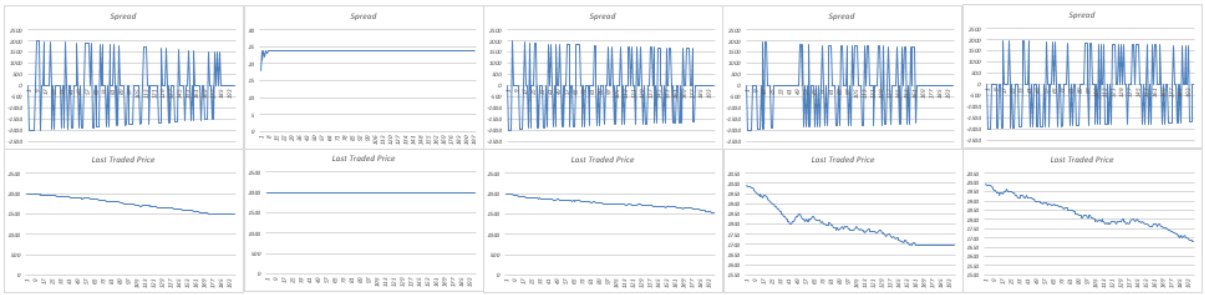


Figure A2-2. The Haskell simulator's output. The first row contains five graphs of Spread, and the second row contain five graphs of Last Traded Price (LTP).

The output Spread and LTP of both the Miranda and Haskell simulator have the similar trends. The ported Haskell simulator is valid.

2.2 Final result verification.

To verify the effect of this project, we made a comparison between the original Miranda simulator and the new Haskell simulator. We picked a typical test – testmeHetero with 5 Hft agents and 200 sim steps. We ran 15 trials on both simulators to compare their performance. Here is the result:

	Time(s)	Miranda	Haskell
Trial 1	Real	47.233	0.295
	User	46.768	0.254
	Sys	0.415	0.015
Trial 2	Real	69.378	0.227
	User	68.968	0.214
	Sys	0.399	0.011
Trial 3	Real	83.693	0.377
	User	83.269	0.36
	Sys	0.415	0.015
Trial 4	Real	51.176	0.257
	User	50.758	0.243
	Sys	0.411	0.012
Trial 5	Real	67.378	0.354
	User	66.963	0.339
	Sys	0.404	0.014
Trial 6	Real	64.94	0.353
	User	64.514	0.336
	Sys	0.418	0.015
Trial 7	Real	87.588	0.378
	User	87.153	0.361
	Sys	0.424	0.014
Trial 8	Real	124.885	0.258
	User	124.436	0.244
	Sys	0.44	0.012
Trial 9	Real	57.885	0.259
	User	57.466	0.246
	Sys	0.41	0.012
Trial 10	Real	53.521	0.377
	User	53.111	0.36
	Sys	0.403	0.015
Trial 11	Real	103.625	0.258
	User	103.187	0.245
	Sys	0.43	0.011

Trial 12	Real	108.811	0.266
	User	108.378	0.253
	Sys	0.428	0.012
Trial 13	Real	113.684	0.382
	User	113.261	0.364
	Sys	0.417	0.015
Trial 14	Real	48.435	0.307
	User	48.024	0.292
	Sys	0.4	0.013
Trial 15	Real	46.497	0.289
	User	46.081	0.275
	Sys	0.41	0.012
Mean	Real	75.2486	0.309133
	User	74.82247	0.2924
	Sys	0.414933	0.0132

Table A2-1, The cost of 15 runs for the simulator in Miranda and Haskell.

We also did a T-test on the real time cost to find out what extent these two distributions of performance overlap. The "p-value" is 3.29494E-08.

3 Code listing[3]

3.1 Comm.hs

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE FlexibleInstances #-}
module Comm where                                   --common types and functions

import Baserands
import Data.Time.Clock
import System.IO.Unsafe

-----------------------------------------------------------------------
--BASIC TYPES

type Price = Double
type Size = Double
type Time = Int
type Inventory = Size

data Ordertype = Bid Limitorder_t | Offer Limitorder_t | Sell Marketorder_t | Buy Marketorder_t | None |
Abort
                        deriving (Show,Eq)
data Marketorder_t = Orkill | Andkill
                            deriving (Show,Eq)
data Limitorder_t = Goodtillcancelled | Goodtilldate Time  -- Not sure where these two should be -
Minimumquantity | Maximumquantity | Day
                           deriving (Show,Eq)
data Traderid = Phantom | Intermediary Int | HFT Int |  FundSeller Int | FundBuyer Int |  Small Int |
Opportunistic Int |
                  SSTaker Int | BSTaker Int | BSMaker Int | SSMaker Int | Maker Int | Taker Int
                  deriving (Show,Eq)
data Sentiment = Calm | Choppy | Ramp | Toxic
                        deriving (Show,Eq)                                  --need to test Eq

type Order = (Price, Size, (Time, Time), Traderid, Ordertype, Int)
type Orderlist = [(Price, [Order])]

data Lob = LOB Bids Offers Buys_waiting Sells_waiting Trades_done Time Orders_received Sentiment
Traderinfo Statstype
                deriving (Eq)

type Lobsummary_t = [Double]
type Lobtrace = [[Char]]
type Bids = Orderlist
type Offers = Orderlist
type Buys_waiting = [Order]
```

[3] The full code listing is provided on the accompanying CD

```
type Sells_waiting = [Order]
type Trades_done = [(Order, Order)]
type Orders_received = [Order]
type Traderinfo = [Traderinfo_item]
type Traderinfo_item = (Traderid, Percent_liquidity_taken, Penalty_applied, Time)     -- time is time penalty was applied
type Percent_liquidity_taken = Double
type Penalty_applied = Time
type Statstype = (Price,Price,Double,Double,Double,Double, --  underlying value, last traded price, squared error, cumulative squared error,samples,percent,
                  (Size,Size,Size,Size),                    --  minliquiditybs, maxliquiditybs, liquiditybs, maxdrawdownbs,
                  (Size,Size,Size,Size),                    --  minliquidityss, maxliquidityss, liquidityss, maxdrawdownss
                  (Size,Size,Size,Size),                    --  minliquiditybstop, maxliquiditybstop, liquiditybstop, maxdrawdownbstop,
                  (Size,Size,Size,Size),                    --  minliquiditysstop, maxliquiditysstop, liquiditysstop, maxdrawdownsstop
                  (Size,Size,Size,Size),                    --  zeroliquiditybs, maxzeroliquiditybs, zeroliquidityss, maxzeroliquidityss
                  (Size,Size,Size,Size)                     --  zeroliquiditybstop, maxzeroliquiditybstop, zeroliquiditysstop, maxzeroliquiditysstop
                 )

data Msg_t = Hiaton | Cancelmessage (Int,Int) (Int,Int) | Message (Int,Int) [Arg_t] | Ordermessage (Int,Int) Order | Trademessage (Int,Int) Order Order| Broadcastmessage (Int,Int) Broadcast_t | Datamessage (Int,Int) [Char] | Ackmessage (Int,Int) Int Order [Char] | Debugmessage (Int,Int) [Char]
           deriving (Show,Eq)     --(from,to) identifiers

data Broadcast_t = Numlistbroadcast [Double] | Strbroadcast Str
                   deriving (Show,Eq)    -- Add your self-defined broadcasts here

data Str = EmptyString | Str [Char]
           deriving (Show,Eq)
--data Arg_t    = EmptyArg | Arg (Str,Int )
data Arg_t    where{
        EmptyArg :: Arg_t;
        Arg :: (Str,Double)-> Arg_t                    -- need more discussion
}     deriving (Show,Eq)
arg_getstr :: Arg_t -> [Char]
arg_getstr (Arg ((Str x),y)) = x
arg_getnum (Arg ((Str x),y)) = y
arg_findval key [] = -1
arg_findval key ((Arg ((Str x),y)):t) =    if x == key
                                           then y
                                           else arg_findval key t
arg_findstr key [] = ""
arg_findstr key ((Arg ((Str x),y)):t) =    if y == key
                                           then x
                                           else arg_findstr key t



type Agent_t          = Agentstate_t -> [Arg_t] -> [(Int, [Msg_t], [Msg_t])] -> Int -> [[Msg_t]]    --Agents take a tuple of (time, messages, broadcasts) last num is id from sim
data Agentstate_t = Agentstate (Double, Double, Int, Lob)
```

```
                              | Emptyagentstate
                              | Exchstate Lob
                              | Nicemimestate Nice_mime_lob
                              | Traderstate (Price, Price, Double, Sentiment, Double, [Order -> Order])
                              | Newagstate ([Double], [Order], Sentiment, [Order -> Order], [Double], [Double])
--traderstate is old_bestbid, old_bestoffer, old_ordernum
                             deriving (Eq)
emptyagentstate = Emptyagentstate          -- we don't yet know what this will or should be
instance Eq (Order -> Order) where
        x == y = True
        x /= y = not (x == y)

data Nice_mime_lob = Nice_mime_lob Sentiment Listoftotals Nubids Nuoffers Time Ticksize Lasttradedprice
Stats
                               deriving (Eq)          -- need to be confirmed
type Listoftotals = [Double]
type Stats = [Double]
type Nubids = [Tick]
type Nuoffers = [Tick]
type Ticksize = Double
type Tick = (Double, [Order])
type Lasttradedprice = Double

-----------------------------------------------------------------------

--TYPE CLASSES
-------------------------------------------------------------------------------
class C_order order where
      emptyorder :: order
      isemptyorder :: order -> Bool
      order_create :: Price -> Size -> Time -> Traderid -> Ordertype -> order
      order_getprice :: order -> Price
      order_setprice :: Price -> order -> order
      order_getuid :: order -> Int
      order_setuid :: [Char] -> Int -> order -> order
      order_gettime :: order -> Time
      order_setfractime :: Time -> order -> order
      order_getfractime :: order -> Time
      order_gettype :: order -> Ordertype
      order_gettraderid :: order -> Traderid
      order_gettraderidno :: order -> Int
      order_getsize :: order -> Size
      order_newsize :: order -> Size -> order
      order_newprice :: order -> Price -> order
      order_equals :: order -> order -> Bool
      order_samesignature :: order -> order -> Bool
      order_getexpiry :: order -> Time
      showorder :: order -> [Char]


class C_Orderlist orderlist where
      emptyorderlist :: orderlist
      isemptyorderlist :: orderlist -> Bool
      ol_first :: orderlist -> (Price,[Order])
      ol_last :: orderlist -> (Price,[Order])
      ol_insert :: Price -> orderlist -> Order -> orderlist
```

```
    ol_delete :: Price -> orderlist -> Order -> orderlist
    ol_reverse :: orderlist -> orderlist
    ol_gethighestprice :: orderlist -> Price
    ol_getlowestprice :: orderlist -> Price
    ol_getliquidity :: orderlist -> Size
    ol_getlevels :: orderlist -> Int                                -- neet to confirm
    ol_getdepth :: Ordertype -> orderlist -> Size                   -- neet to confirm
    ol_getdepthneartop :: Ordertype -> Double -> orderlist -> Size      -- neet to confirm
    ol_poplowest :: orderlist -> (Order, orderlist)
    ol_clean :: orderlist -> orderlist
    ol_fill :: orderlist -> Order -> Ordertype -> [(Order, Order)] -> (orderlist, Order, [(Order, Order)])
    ol_fill_cross :: orderlist -> Order -> Ordertype -> [(Order, Order)] -> (orderlist, Order, [(Order, Order)])
    showorderlist :: orderlist -> [Char]


class C_Lob lob where
    emptylob :: lob
    primed_emptylob :: lob
    testlob :: Int -> lob
    lob_increment_time :: lob -> lob
    lob_clear_trades :: lob -> lob
    lob_updatestats :: lob -> lob
    lob_setpercent :: Double -> lob -> lob
    lob_getpercent :: lob -> Double
    lob_gettime :: lob -> Time
    lob_getbestbid :: lob -> Price
    lob_getbestoffer :: lob -> Price
    lob_getsentiment :: lob -> Sentiment
    lob_setsentiment :: Sentiment -> lob -> lob
    lob_getbuysideliquidity :: lob -> Size
    lob_getsellsideliquidity :: lob -> Size
    lob_getmidprice :: lob -> Price
    lob_getlasttradedprice :: lob -> Price
    lob_getordernum :: lob -> Int
    lob_getbuysidelevels :: lob -> Int
    lob_getbuysidedepth :: lob -> Size
    lob_getbuysidedepthneartop :: lob -> Size
    lob_getsellsidelevels :: lob -> Int
    lob_getsellsidedepth :: lob -> Size
    lob_getsellsidedepthneartop :: lob -> Size
    lob_gettraderinfo :: Traderid -> lob -> Traderinfo_item
    lob_settraderinfo :: Traderinfo_item -> lob -> lob
    lob_gettrace :: lob -> [Char]
    lob_getstats :: lob -> Statstype
    newbid :: Price -> Order -> lob -> lob
    newoffer :: Price -> Order -> lob -> lob
    newtrade :: Price -> Order -> lob -> lob
    neworder :: Order -> lob -> lob
    lob_gettrades :: lob -> Trades_done
    execute_trades :: lob -> lob
    is_crossed_book :: lob -> Bool
    uncross_book :: lob -> lob
    match :: Order -> lob -> lob
    match_m1 :: Order -> lob -> lob
    match_m2 :: Order -> lob -> lob
    match_m3 :: Order -> lob -> lob
```

```
        match_m4 :: (Double,Double,Double,Double,Double) -> Order -> lob -> lob
        showlob :: lob -> [Char]


class C_msg_t msg_t where
        hiaton :: msg_t
        msg_getid :: msg_t -> Int
        msg_getfromid :: msg_t -> Int
        showmsg_t :: msg_t -> [Char]
        message :: (Int,Int) -> [Arg_t] -> msg_t
        ordermessage :: (Int,Int) -> Order -> msg_t
        cancelmessage :: (Int,Int) -> (Int,Int) -> msg_t -- First tuple is from/to, second tuple is tid/ouid
        trademessage :: (Int,Int) -> Order -> Order -> msg_t
        datamessage :: (Int, Int) -> [Char] -> msg_t
        debugmessage :: (Int,Int) -> [Char] -> msg_t
        ackmessage :: (Int, Int) -> Int -> Order -> [Char] -> msg_t --0 accept, 1 order too large, 2 no more liquidity,
3 outside sliding window of acceptable prices, 4 too many contracts on book, 5 order cancelled, 6 book is
spiked, 7 minimum resting time not obeyed.
        broadcastmessage :: (Int,Int) -> Broadcast_t -> msg_t
        msg_isbroadcast :: msg_t -> Bool
        msg_isdata :: msg_t -> Bool
        msg_disptrace :: msg_t -> [Char]
        msg_getbroadcast :: msg_t -> Broadcast_t
        msg_getorders :: [msg_t] -> [Order] -- returns orders from list of messages, ignores non ordermessages
        msg_getnumlistfrombcast :: msg_t -> [Double]
        msg_gettrade :: msg_t -> [Order]
        msg_isack :: msg_t -> Bool
        msg_getackcode :: msg_t -> Int
        msg_istrade :: msg_t -> Bool
        msg_isreject :: msg_t -> Bool    --Named this isreject instead of isack to avoid nameclashing and
confusion.
        msg_getcanceltuple :: msg_t -> [(Int,Int)]
        msg_getackdorder :: msg_t -> Order


class C_broadcast_t broadcast_t where
        showbroadcast_t :: broadcast_t -> [Char]
        broadcast_getnumlist :: broadcast_t -> [Double]
        broadcast_numlist :: [Double] -> broadcast_t
        broadcast_str    :: Str -> broadcast_t

------------------------------------------------------------------------------


------------------------------------------------------------------------------
--DISTRIBUTIONS

--rands generates random numbers in the range 0 - 1,000
randoms :: [Int]
randoms = baserands ++ randoms

gaussians :: [Double]
gaussians = mygaussians (map fromIntegral randoms) -- range is -6000 to 6000
                where
                mygaussians [] = []
                mygaussians (a:b:c:d:e:f:g:h:i:j:k:l:rest) = ((a+b+c+d+e+f+g+h+i+j+k+l) - 6000):(mygaussians rest)
```

```
-- value is a function that gives the underlying (fundamental) value of the stock being traded as a function of time.    It can be uniform
-- (the easiest case) or rising, falling or sinusoidal.

value:: Sentiment->Time->Double
value Calm      t = 2000
value Choppy t = ((sines !!t) + 1.0    ) * 1500 + 1000
value Ramp      t = ([2000,1999 ..]!! t)
value Toxic    t = if t < 40
                          then 2000
                          else 5

sines:: [Double]
sines = [sin x | x <- [0.0,0.1 .. (2*pi)]] ++ sines

-------------------------------------------------------------------------------
--Helper function
rep n x = take n (repeat x)

systemTime :: Num a=> a -> Double                                        --need to verify
systemTime x = unsafePerformIO $ getTime
                     where
                     getTime = getCurrentTime >>= return . fromRational . toRational .utctDayTime     --using
monad

cpsafehd caller x = safehd x caller
safehd :: [a] -> [Char] -> a
safehd x caller = if not(null x)
                          then head x
                          else error ("safehd - from " ++ caller)

zipfunc x [] = []
zipfunc [] x = []
zipfunc (f:rf) (a:ra) = ((f a) : (zipfunc rf ra))
si warn list n = if n < (length list)
                          then    list!!n
                          else error warn

mysuperor f1 f2 x = or [(f1 x), (f2 x)]

mymod:: Double -> Double -> Double                    -- be careful of the type
mymod x 0 = error "mymod applied to zero"
mymod x y    | y>x    = x
                | otherwise = x - (fromIntegral(floor(x/y)))*y

--mymod x y |(x >= 2*y)       = mymod (x-y) y       --recursive version
--                |(y > x)             = x
--                |otherwise         = x-y

-------------------------------------------------------------------------
```

3.2 Sim.hs

```
module Sim where
```

```
import Comm
import Stats
import Messages
import System.IO
import Data.List
import Data.Char

import Control.Monad.Par

assetreturnscoln = 27
--The main simulator
---------------------------------
--Args are (string,value) pairs with numeric values - these are passed on to simstep, sim_updatestate and agents
--The "agents" parameter to sim is a list of functions, which are applied to their arguments inside sim
--Once applied to their full set of arguments, agents consume the potentially-infinite stream of simulator system states ("allstates")
--and each produces a potentially-infinite stream of lists of messages to be sent to other agents (according to the target id in the message).
--Notice that at each timestep an agent can produce a list of messages - because one agent might want to simultaneously send messages to several different
--other agents.
--They do this in a recursive loop - it is important that (i) there is a precomputed first state for agents to view, (ii) the
--agents only ever look at the head of the incoming list of states each time around their own loop, and (iii) the "simstep" function
--only ever looks at the head of the message lists from the agents each time around its own loop.  Any attempt to lookahead will cause the simulation
--to deadlock.
--Any message sent to target id "0" is a message for the simulator harness and is used to update the simulator state - which is
--printed to the trace file and is also visible to all other agents.
--Each agent has its own private state.
--The last arg to sim is the required highest group number for broadcast messages

sim :: Int -> [Arg_t] -> [(Agent_t, [Int])] -> IO()
sim steps args agents = do
                        trace_outh<-openFile (prefix ++ "trace") WriteMode
                        data_outh<- openFile (prefix ++ "data.csv") WriteMode
                        hSetBuffering trace_outh (BlockBuffering Nothing)         -- set buffer mode
                        hSetBuffering data_outh    (BlockBuffering Nothing)         -- set buffer mode

                        --start output
                        hPutStr trace_outh ("START OF SIMULATION - seed is " ++ seedmsg ++ "\nNB System Time 2 aligns with Exchange Time 1\n\n")      --need to confirm the recursive behaviour
                        tracer allstates trace_outh data_outh
                        statstuff

                        where
                        statstuff = if (arg_findval "autostats" args) /= -1
                                    then    (printstatoutput    (statfuncs    (prefix    ++    "data.csv") assetreturnscoln True) (prefix ++ "-asset_returns-"))
                                    else    return()      --do nothing
                        highestbrg = maximum ((concat (map snd agents)) ++ [0]) --The zero is incase there are no agents subscribed to any broadcast groups.
```

```
                    myrands = if (arg_findval "randseed" args) /= -1
                                then drop (pmh + (length agents)) randoms
                                else drop ((round (arg_findval "randseed" args)) * (pmh + (length
agents))) randoms
                    seedmsg = if (arg_findval "randseed" args) == (-1)
                                then "default"
                                else show (arg_findval "randseed" args)
                    pmh = (sum (map ord prefix)) --Poor Man's Hash (function)'
                    prefix = if (arg_findstr 9989793425 args) /= ""
                              then (arg_findstr 9989793425 args) ++ "-"
                              else    ""
                    startsimstate = sim_emptystate agents highestbrg myrands
                    allstates = startsimstate : simstates
                    simstates = simstep steps 1 args startsimstate allmessages myrands
                    allmessages = map f (zip [1..] agents)
                                  where
                                  -- f is a function that fetches the messages and broadcasts for
a given agent (using sim_getmymessages and sim_getmybroadcasts)
                                  -- and then applies the agent to (i) the empty state; (ii) the
args (that are sent to the simulator at the start);
                                  -- (iii) an infinite list of lists of messages and broadcasts, and
(iv) the agent id.
                                  -- It fetches the messages trivially by using the agent's ID to
index into the message list
                                  -- (which is sorted by destination ID for each message).
                                  -- It fetches the broadcasts for an agent as follows:
                                  -- - each agent is subscribed to a list of broadcast groups (this
is the data called brcs)
                                  -- - for each subscribed broadcast group ID, that ID is used to
index into the broadcast list (which is
                                  --    held in the state called st, and which is sorted into
broadcast groups).    The indexing is done by sim_getmybroadcasts.
                                  -- - all of the returned broadcast lists (for different broadcast
group IDs) are concatenated into a single list.
                                  f :: (Int,(Agent_t, [Int]))    -> [[Msg_t]]
                                  f (id,(a, brcs)) = a emptyagentstate args (map (g id) allstates)
id
                                                     where
                                                     g x st = ((fromIntegral $sim_gettime
st), sim_getmymessages st x, concat (map (sim_getmybroadcasts st) brcs))

                    simstep      n      t      args      st      []                myrnds                =      []
-- for completeness - should never happen
                    simstep      0      t      args      st      msgs      myrnds                =      []
-- end of simulation steps
                    simstep n t args st msgs myrnds      = if (elem msgs [])
                                                     then                                  []
-- end simulation if any agent ends
                                                     else newstate : (simstep (n-1) (t+1)
args newstate (map tail msgs) (drop t myrnds))
                                                     where
                                                     newstate = sim_updatestate t args
cleanst (map (cpsafehd "simstep") msgs) myrnds    -- t is the "time" - the simulation step
                                                     cleanst = sim_clean_msg_br st
```

```
--Tracer can take an argument to send its output to specified files. The argument is detailed below.
--Presence of (Arg (String prefix, 9989793425)) indicates files should be written to the messyoutputfolder folder in the files prefixed by the specified prefix.
--E.g. if prefix is "MyLatestTest" files will be written to folder "MyLatestTest" and the files will be called "MyLatestTest-data.csv" and "MyLatestTest-trace".
--The number 9989793425 acts as a key so that the HFT agent can find the value (the prefix).

tracer :: [Simstate_t] -> Handle->Handle -> IO()
tracer [] traceh datah    = do
                                    hPutStr traceh ("END OF SIMULATION\n")

                                    hClose traceh
                                    hClose datah

tracer (x:xs) traceh datah = do
                                       hPutStr traceh ((showsimstate_t x) ++ "\n==============\n") --showsimstate_t ?
                                       datatraces
                                       tracer xs traceh datah
                                       where
                                       datatraces = hPutStr datah dtracepostformatting
                                       dtracepostformatting = if dtracepreformatting /= ""
                                                                  then (show (sim_gettime x)) ++ ", " ++ dtracepreformatting ++ "\n"       -- time added for debugging CDC 8/4/2014
                                                                  else (show (sim_gettime x)) ++ ", " ++ dtracepreformatting
                                       dtracepreformatting = ((concat (map msg_disptrace (filter msg_isdata (sim_getmymessages x 0)))))



--The abstract type definition for simulator state
-------------------------------------------------------------------

--abstype simstate_t
--with
--      sim_emptystate :: [(agent_t,[num])] -> num -> [num] -> simstate_t                                                    || creates an empty message stream for each agent
--      sim_updatestate :: num -> [arg_t] -> simstate_t -> [[msg_t]] -> [num] -> simstate_t     || time, args, oldst, one msg_t per agent
--      showsimstate_t :: simstate_t -> [char]
--      sim_gettime :: simstate_t -> num
--      sim_getmymessages :: simstate_t -> num -> [msg_t] ||num is id
--      sim_getmybroadcasts :: simstate_t -> num -> [msg_t] ||num is groupid
--      sim_clean_msg_br :: simstate_t -> simstate_t
--      || and so on - we will need access methods
--

--The implementation of simulator state
--It must implement methods sim_emptystate, sim_updatestate and showsimstate_t

type Simstate_t = ([[Msg_t]], Int, [Msg_t], [[Msg_t]])
                                                -- list of messages for each agent - a snapshot at time t - plus time and other info
```

```
sim_emptystate :: [(Agent_t,[Int])] -> Int -> [Int] -> Simstate_t
sim_emptystate []        hbrg rnds = ([[]], 0, [(debugmessage (0,0) (show (take 30 rnds)))], (map f [0..hbrg]))
-- this base case gives the extra list of messages for agent id which is the sim harness
                                        where
                                        f x = []
sim_emptystate (a:as) hbrg rnds = (([]:x), b, c, br)
                                        where
                                        (x,b,c,br) = sim_emptystate as hbrg rnds

sim_clean_msg_br :: Simstate_t -> Simstate_t
sim_clean_msg_br (m, b, c, br) = (emptym, b, c, emptybr)
                                    where
                                    emptym = map f m
                                    emptybr = map f br
                                    f x = []

sim_updatestate :: Int -> [Arg_t] -> Simstate_t -> [[Msg_t]] -> [Int] -> Simstate_t
sim_updatestate t args (m, b, c, br) [] myrands        = (newm, t, safehd newm "sim_updatestate", reverse br2)
--just copy over broadcast??
                                                        where
                                                        newm = reverse (map reverse m2)
                                                        m2 = if randomise == True
                                                              then ((map (randomWrap myrands)
(take ((length m) - 1) m)) ++ (drop ((length m) - 1) m))
                                                              else m
                                                        br2 = if randomise == True
                                                               then (map (randomWrap myrands)
br)
                                                               else br
                                                        randomise = if (arg_findval "Randomise"
args) /= (-1)
                                                                    then True
                                                                    else False
sim_updatestate t args (m, b, c, br) (x:xs) myrands
                                                    = sim_updatestate t args (fm, b, c, fcasts) xs (drop 72
myrands)
                                                      where
                                                      --updatemsgs takes messages from agents and puts
them in msgqueues for agents (id 0 = for sim)
                                                      updatemsgs 0 (y:ys) = [(filter ((==0).msg_getid) (filter
((not).msg_isbroadcast) x)) ++ y]
                                                      updatemsgs n (y:ys) = ((filter ((==n).msg_getid) (filter
((not).msg_isbroadcast) x)) ++ y): (updatemsgs (n-1) ys)
                                                      updatecasts 0 (y:ys) = [(filter ((==0).msg_getid) (filter
msg_isbroadcast x)) ++ y]
                                                      updatecasts n (y:ys) = ((filter ((==n).msg_getid) (filter
msg_isbroadcast x)) ++ y): (updatecasts (n-1) ys)
                                                      newm =    (updatemsgs ((length m) - 1) ( m))
                                                      newcasts = (updatecasts ((length br) - 1) ( br))
--These guys need to know the number of broadcast groups...
                                                      --        (fm, fcasts) = (((map (randomWrap myrands)
(take newml newm)) ++ (drop newml newm)),(map (randomWrap myrands) newcasts)), if randomise = True
                                                      (fm, fcasts) = (newm, newcasts)
                                                      -- newml = (# newm) - 1
                                                      -- randomise = True, if (arg_findval "Randomise" args)
~= (-1)
```

```
                                                --                  = False, otherwise
                                                --

showsimstate_t :: Simstate_t -> [Char]
--showsimstate_t (m,b,c,br) = runPar $do            -- static partitioning,one task for each element in m
--                                  let t = show b
--                                  let bm = (concat (map (concat.(map showmsg_t)) (drop 1 br)))
--                                  let h    = (concat (map showmsg_t c))
--                                  msg2Sim <- parMap (concat.(map showmsg_t)) (drop 1 m)
--                                  let m2s = concat msg2Sim
--                                    return (", \n\nSystem Time: "++t++"\nMessages to sim: "++m2s++",
\n\nHarness messages: " ++ h ++ "\n\nBroadcasts: "++bm++"\n\n\nEND OF STATE\n")

--showsimstate_t (m,b,c,br) = runPar $do            -- static partitioning, split in half
--                                  let t = show b
--                                  let bm = (concat (map (concat.(map showmsg_t)) (drop 1 br)))
--                                  let h    = (concat (map showmsg_t c))
--                                  let mm = drop 1 m
--                                  --i1 <-new
--                                  --i2 <-new
--                                  --fork $ put i1 (map showmsg_t (head mm))
--                                  --fork $ put i2 (map (concat.(map showmsg_t)) (tail mm))
--                                  i1 <-    (spawn.return.(map showmsg_t)) (head mm)
--                                  i2 <-    (spawn.return.(map (concat.(map showmsg_t)))) (tail mm)
--                                  m11 <- get i1
--                                  m22 <- get i2
--                                  let m2s = concat (m11 ++ m22)
--                                    return (", \n\nSystem Time: "++t++"\nMessages to sim: "++m2s++",
\n\nHarness messages: " ++ h ++ "\n\nBroadcasts: "++bm++"\n\n\nEND OF STATE\n")



--showsimstate_t (m,b,c,br) = runPar $do       -- straightforward solution
--                                  sysT       <- spawnP $ show b
--                                  msg2Sim <- spawnP $ (concat (map (concat.(map showmsg_t)) (drop 1
m)))
--                                  hMsg       <- spawnP $ (concat (map showmsg_t c))
--                                  bMsg       <- spawnP $ (concat (map (concat.(map showmsg_t)) (drop 1
br)))
--                                  t <- get sysT
--                                  m2s <- get msg2Sim
--                                  h <- get hMsg
--                                  bm <- get bMsg
--                                    return (", \n\nSystem Time: "++t++"\nMessages to sim: "++m2s++",
\n\nHarness messages: " ++ h ++ "\n\nBroadcasts: "++bm++"\n\n\nEND OF STATE\n")

showsimstate_t (m,b,c,br)    = ", \n\nSystem Time: "++t++"\nMessages to sim: "++m2s++", \n\nHarness
messages: " ++ h ++ "\n\nBroadcasts: "++bm++"\n\n\nEND OF STATE\n"    --for profiling
                                  where
                                  t = show b
                                  bm = (concat (map (concat.(map showmsg_t)) (drop 1 br)))
                                  h    = (concat (map showmsg_t c))
                                  m2s = concat msg2Sim
                                  msg2Sim = [f] ++ g
                                            where
                                            f = (concat.(map showmsg_t)) (head (drop 1 m))
```

```
                                        g = map (concat.(map showmsg_t)) (tail (drop 1 m))



--showsimstate_t (m,b,c,br) = ", \n\nSystem Time: "++(show b)++"\nMessages to sim: "++(concat (map (concat.(map showmsg_t)) (drop 1 m)))++", \n\nHarness messages: " ++ (concat (map showmsg_t c)) ++ "\n\nBroadcasts: "++(concat (map (concat.(map showmsg_t)) (drop 1 br)))++"\n\n\nEND OF STATE\n"
--original

--Dropping 1 from m to avoid printing harness messages twice.
--showsimstate_t (m,b,c,br) = ", \n\nTime: "++(show b)++"\nMessages to sim: "++(concat (map (concat.(map show)) m))++", \n\nUnknown: "++(show c)++", \n\nBroadcasts: "++(show br)++"\n\n\nEND OF STATE\n" OLD UNKNOWN FIX ME
--Only index 2 of messages and br is populated and for some reason only one message is retained...

sim_gettime :: Simstate_t -> Int
sim_gettime (m, b, c, br) = b

sim_getmymessages :: Simstate_t -> Int -> [Msg_t]
sim_getmymessages (m,b,c,br) idnum = si "sim:1" m idnum

sim_getmybroadcasts :: Simstate_t -> Int -> [Msg_t]
sim_getmybroadcasts (m,b,c,br) groupnum = si "sim:2" br groupnum

--Helper functions
-------------------------------------------
randomWrap myrands list
                        = randomize (map ((* coef).(converse (-) 500)) myrands) list
                          where
                          coef = if len < 500
                                    then 1
                                    else (floor (len / 500)) + 1        --entier== floor
                          len = fromIntegral $length list

randomize rands [] = []
randomize (r:rs) msgs
  =    a : (randomize rs b)
       where
       a = msgs !! ( mod r len)
       len = length msgs
       b = listsub msgs    [a]


listsub :: Eq a=>[a]->[a]->[a]

listsub x [] = x
listsub x (b:y) =    listsub (delete b x) y
```

3.3 Message.hs

```
module Messages where
```

```
import Order
import Comm



--The message type & broadcast message type
---------------------------------

instance C_msg_t Msg_t    where
      hiaton = Hiaton

      --constructors.
      message x args    = Message x args
      debugmessage x y = Debugmessage x y
      ordermessage x anOrder = Ordermessage x anOrder
      broadcastmessage x broadcast = Broadcastmessage x broadcast
      cancelmessage fromto idtuple = Cancelmessage fromto idtuple
      trademessage fromto ordera orderb = Trademessage fromto ordera orderb
      datamessage from dat = Datamessage from dat
      ackmessage fromto ackd x msgg = (Ackmessage fromto ackd x msgg)

      --msg_t functions
      msg_getid Hiaton = 0
      msg_getid (Message (from,to) args) = to
      msg_getid (Ordermessage (from,to) args) = to
      msg_getid (Broadcastmessage (from,to) args) = to
      msg_getid (Trademessage (from,to) ordera orderb) = to
      msg_getid (Datamessage (from, to) dat) = to
      msg_getid (Ackmessage (d,e) b c m) = e
      msg_getid (Cancelmessage (f,t) b) = t
      msg_getid (Debugmessage (f,t) m) = t
      msg_getfromid Hiaton = 0
      msg_getfromid (Message (from,to) args) = from
      msg_getfromid (Ordermessage (from,to) args) = from
      msg_getfromid (Broadcastmessage (from,to) args) = from
      msg_getfromid (Trademessage (from,to) ordera orderb) = from
      msg_getfromid (Datamessage (from, to) dat) = from
      msg_getfromid (Ackmessage (d,e) b c m) = d
      msg_getfromid (Cancelmessage (f,t) b) = f
      showmsg_t Hiaton = "\nHiaton"
      showmsg_t (Datamessage x dat) = ""
      showmsg_t (Cancelmessage x (trader, order)) = "\nMessage from/to " ++ (show x) ++ ": Trader #" ++
(show trader) ++ " is cancelling order " ++ (show order)
      showmsg_t (Message x args) = "\nMessage from/to "++(show x)++" ["++(concat (map arg_getstr args)) ++
"]\n"
      showmsg_t (Ordermessage x args) = "\nMessage from/to "++(show x)++" "++(showorder args) ++ ""
      showmsg_t (Broadcastmessage x args) = "\nBroadcast from/to group "++(show x)++"
["++(showbroadcast_t args) ++ "]"
      showmsg_t (Trademessage (from,to) ordera orderb) = "\n==========\n\nTo: " ++ (show to) ++ "\nOrder:
\n" ++ (show ordera) ++ "\n\n matched with\n\nOrder:" ++ (show orderb) ++ "\n\n=========="
      showmsg_t (Ackmessage (d,e) b c m) = "\nAckmessage(" ++ (show d) ++ "->" ++ (show e) ++ "): The
following order " ++ text ++ (show c)
                                          where
                                          text| b /= 0          = "was unsuccessful because " ++ m ++ ":
\n"
```

```
                                                    | b == 5            = "was cancelled successfully."
                                                    | otherwise        = "was successful: \n"
        showmsg_t (Debugmessage ft m) = "\n+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=\nDebug message:" ++ (show ft) ++
"\n" ++ m ++ "\n+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+="
        msg_disptrace (Datamessage x dat) = dat
        msg_disptrace any = ""


        msg_getnumlistfrombcast (Broadcastmessage a b) = broadcast_getnumlist b
        msg_isbroadcast (Broadcastmessage a b) = True
        msg_isbroadcast any = False
        msg_isdata (Datamessage a b) = True
        msg_isdata any = False
        msg_isack (Ackmessage a b c m) = True
        msg_isack any = False
        msg_getackcode (Ackmessage a b c m) = b
        msg_getackcode any = error "msg_getackcode - calling get ack code on non-ack message."
        msg_getackdorder (Ackmessage a b c m) = c
        msg_getackdorder any = error "msg-getackdorder - Called this on a non-ack message"
        msg_istrade (Trademessage (from,to) ordera orderb) = True
        msg_istrade any = False
        msg_isreject (Ackmessage a b c m) = b /= 0
        msg_isreject any = error "msg_isreject called on non-ack message."
        msg_getbroadcast (Broadcastmessage a b) = b
        msg_getbroadcast any = error "Make sure you're trying to get the broadcast FROM a broadcast message
first..."
        msg_getorders [] = []
        msg_getorders ((Ordermessage a b) : rest) = b : (msg_getorders rest)
        msg_getorders (any:rest) = msg_getorders rest
        msg_gettrade (Trademessage (from,to) ordera orderb) = [ordera, orderb]
        msg_gettrade any = []
        msg_getcanceltuple (Cancelmessage x b) = [b]
        msg_getcanceltuple any = []


--The implementation of broadcasts
instance C_broadcast_t Broadcast_t    where
        showbroadcast_t (Numlistbroadcast nums) = "LOBbcast contents: ["++(show nums) ++ "]"
        showbroadcast_t (Strbroadcast string) = show string
        broadcast_getnumlist (Numlistbroadcast lobsum) = lobsum

        broadcast_numlist ob = Numlistbroadcast ob
        broadcast_str string = Strbroadcast string
```