Nizar Noorani
CS379H
12/20/2001

University of Texas At Austin
Department of Computer Science

Comparison of a Simulation System in Haskell vs. Java

Reading and Research - CS379H
Nizar Noorani
Instructor: Hamilton Richards

Nizar Noorani
CS379H
12/20/2001

CONTENTS

Nizar Noorani
CS379H
12/20/2001

Abstract:

This paper examines the differences involved in implementing a program in an imperative language versus a pure functional language and discusses the advantages and disadvantages of these differences.

In particular, this paper discusses the design of a simulation system and its implementation in the Object-Oriented language Java and the pure functional language Haskell. It will then examine the two implementations and make appropriate conclusions.

Introduction:

The simulation system designed and implemented is a fast-food restaurant, called RestaurantSim. As we will see below, the activities involved in a fast-food restaurant make the restaurant an excellent candidate to be implemented as a simulation system.

Description of RestaurantSim:

RestaurantSim is responsible for simulating the events and activities that occur at a fast-food restaurant. Some of the events that take place at RestaurantSim are the following:

1. Customer arrives at the restaurant
2. Customer leaves the restaurant without ordering
3. Customer leaves the restaurant with the order

Some of the activities that occur at RestaurantSim are:

1. A cashier serves a customer
2. A cook assembles an order
3. A grill is used to cook the meals in an order

In order to perform these activities and events, ResturantSim needs employees and inventory. It, therefore, contains the following employees:

- Cashiers who will serve customers
- Cooks who will assemble orders to be cooked by a grill.

It also contains the following inventory:

- Grills that will be used in cooking the meals
- Meals.

Nizar Noorani
CS379H
12/20/2001

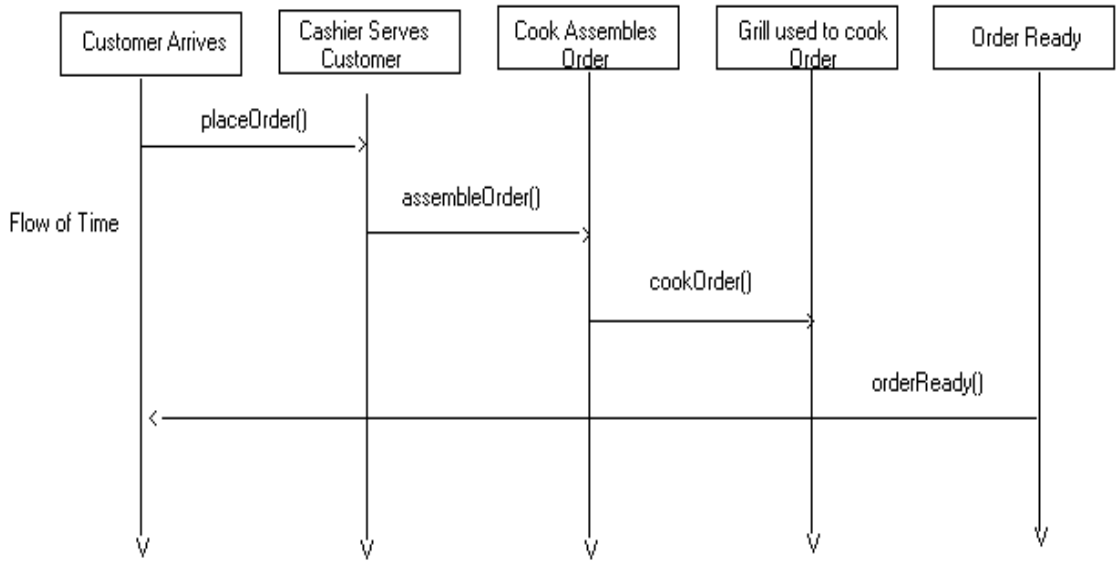
Design of RestaurantSim:

Let us now consider how we might implement a system that adheres to the description given above.

First note that the activities of RestaurantSim are performed in a sequential order with respect to a specific event. For instance, a cashier serves a customer only after the customer has entered the store. Another observation is that some amount of time elapses between the occurrences of two events in which one follows the other.

Below is a sequential diagram that depicts the flow of time and the sequential order in which activities are performed at RestaurantSim with respect to a customer.

RestaurantSim
Sequential Diagram



It is also possible that two customers might arrive the restaurant at the same time. If only one cashier is available, then this would require one customer to wait while the other is being served. If two cashiers are available then the two customers will be served concurrently. Thus with each activity there is a time associated that specifies when the activity will occur.

In the light of these observations, one possible approach is to implement the restaurant as a simulation system of discrete events. Before we go further with our discussion of RestaurantSim, let's first describe what a simulation system is and how it is used.

Nizar Noorani
CS379H
12/20/2001

Simulation systems and their use

Simulation systems are used to model many real world systems, including business, economic, social, biological, physical, or chemical systems.

The idea behind a simulation system is to describe the behavior of an activity through the use of mathematical or logical models. These models can then be implemented using a computer.

There are two major types of simulation systems - Discrete and Continuous systems. A system is regarded as a continuous or a discrete system, depending upon the way that it changes from one state to another. In a continuous system, the values for a variable maybe continuous functions of time; whereas in a discrete system, the value for a variable may change only at discrete instants.

RestaurantSim is an example of a discrete system. This is because it changes state only at discrete instants. In particular, it changes state only at the start and end of an activity. For example, the entrance of a customer marks the start of an activity. At this point the restaurant enters a new state that reflects the fact that the restaurant now has a customer waiting.

The way discrete systems are implemented is through the use of events, where a time is specified with the occurrence of each event. The occurrence of an event then causes other events to be scheduled giving them their corresponding times of occurrence.

Design of RestaurantSim

In our case, the start of each activity is marked by an event. Each event has a time that specifies when it is to occur and the action that it is to perform. Following are the different events that occur at RestaurantSim:

Event Type	Cause of Creation	Action
1. Customer Arrival	Entrance of a customer.	Creates a Serve Customer event if a cashier is available. If not, then enqueue's the customer in a waiting queue. It also creates a Customer Leaves event, which allows a customer to leave the restaurant without ordering.

2. Serve Customer	Customer needs service from a cashier and a cashier is available for service.	Creates an Order Selected event and a Cashier Free event.
3. Customer Leaves	A customer has the option to leave while he/she is waiting to be served by a cashier. A probability that a customer will leave increases with respect to time.	Checks to see if the customer is still waiting to be served by a cashier. If so, then another Customer Leaves event is created.
4. Order Selected	Customer has placed an order. This also implies that a cashier has finished serving the customer.	Creates an Assemble Order event if a cook is available. If not, then enqueues the order in a waiting queue.
5. Cashier Free	A cashier becomes free. A cashier is free when it is not serving a customer.	Checks to see if any customers are waiting for service. If so, it dequeues them and creates a Serve Customer event.
6. Assemble Order	An order needs to be assembled by a cook and a cook is available for service.	Creates an Order Assembled event and a Cook Free event.
7. Order Assembled	A cook has assembled an order. This also implies that the cook has become free.	Creates a Cook Order event if a grill is available. If not, then enqueues the order in a waiting queue.
8. Cook Free	A cook becomes free. A cook is free when it is not assembling an order.	Checks to see if any orders are waiting to be assembled. If so, it dequeues them and creates an Assemble Order event.
9. Cook Order	An order needs to be cooked using a grill and a grill is available for service.	Creates an Order Ready event and a Grill Free event.
10. Order Ready	An order is ready to be delivered to the customer. This also implies that the grill has become free.	No new events created.
11. Grill Free	A grill becomes free. A grill is free when it is not being used to cook an order.	Checks to see if any orders are waiting to be cooked. If so, it dequeues them and creates a Cook Order event.

12. Hire Employee	An employee is hired.	Creates Cashier, Cook or Grill Free event depending on the type of the employee being hired. Adds the employee to the existing list of employees.
13. Fire Employee	An employee is fired.	Sets the employee's status to fired.

The occurrence of an event can create other events. Those events can then create even more events. For example, the occurrence of a Customer Arrival event might create a Serve Customer event, giving it a time of occurrence. Upon occurring, the Serve Customer event will create two more events - Order Selected and Cashier Free event, giving them their corresponding time of occurrence. It is through these series of creations and occurrences of events that RestaurantSim operates.

Upon creation, all events get enqueued in a global priority queue shared by all events. The events are prioritized by their time of occurrence. The restaurant's central operation consists of dequeuing events form this global queue and asking them to perform their actions. The restaurant stops operation when the time of an event is greater then the restaurant's stop time.

Another issue that we need to address is what happens if two customers enter the restaurant at the same time but there is only one cashier available. In this case, one customer will have to wait while the other is being served. Thus a waiting queue is needed where customers will be enqueued in the order they arrive. Similarly, waiting queues are also required for orders that need to be assembled by cooks and for orders whose meals need to be cooked using grills.

Now that we have a procedure of implementing the various activities of the restaurant, let's consider the following question - How do we specify the number employees and the amount of inventory that the restaurant will contain? Our approach is to read this information from a text file. The text-file will contain the following information in the format specified below:

Start-time Stop-time
Total customers
Employee-type Start-time Stop-time Hourly-rate
..
..
..
Grill Start-time Stop-time
..
..
..
Meal Meal-type cost-price sale-price cook-time number-in-advance
Description of Meal
Meal ....
..
...
...

End of file.

The Start-time and Stop-time refer to time when restaurant starts and ends its operation. Total customers specifies the total number of customers that will enter the restaurant during its operation. Employee-type is the type of the Employee - cashier or cook. For each employee, its start time, end time and hourly rate that he/she is paid, should also be specified. There is no limit to how many employees the file can contain. Following the employees, the input file should specify the number of grills to be installed in the restaurant with their corresponding installation and uninstallation times. There is no limit to how many grills the file can contain. Next, the input file should specify the type of meals that the restaurant offers to its customers. Associated with each meal is its cost price, sale price, the time required to cook it, the number of meals the restaurant can cook in advance, and a description of the meal itself. There is no limit to how many meal-types the file can contain.

Upon the end of its operation, the restaurant will print its output in a file specified by the user. The output file will contain the following information:

Sales and profit information:

1. Total sales:
2. Total cost:
3. Net profit:

Statistical Information:

1. Average customer interarrival time:

2. Employee Utilizations:
   1.Utilization for cashiers
   - Utilization for cashier #1
   - ..
   - ..

   Average utilization for all cashiers:
   2.Utilization for cooks
   - Utilization for cook #1
   - ..
   - ..

   Average utilization for all cooks:
   3.Utilization for grills
   - Utilization for grill #1
   - ..
   - ..

   Average utilization for all grills:

3. Average waiting times:
   1. Average waiting time for a cashier:
   2. Average waiting time for a cook:
   3. Average waiting time for a grill:
   4. Average wait for Customers:

3.Total customers that left:

4.Meal sold:
   1. Number of Meal type #1 sold:
   2. ..
   3. ..

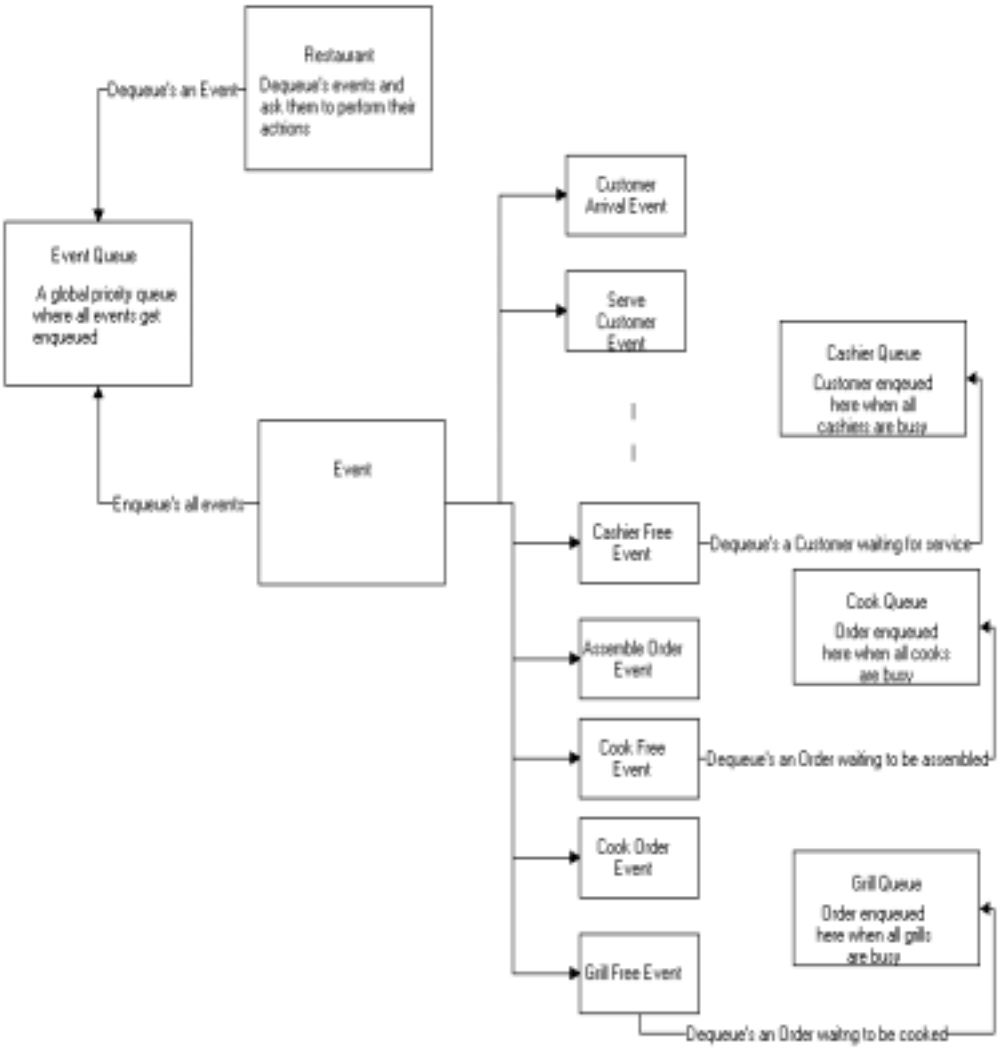
Total meals sold:
Total meals cooked:

End of statistics.

This concludes our discussion of designing RestaurantSim. Below is a diagram that depicts the overall design of RestaurantSim.

Nizar Noorani
CS379H
12/20/2001

RestaurantSim
Overall Design



Note that not all events have been depicted in the diagram whose purpose is to give an overall view of the design.

Now that we have a well-formed design of RestaurantSim, we can start to implement it in a programming language. I will discuss the implementation of RestaurantSim in Java, a modern imperative language, and then in Haskell, a pure functional language.

Implementation in Java

An Object-Oriented approach will be adopted in implementing RestaurantSim. RestaurantSim will contain the following objects:

- Event objects that implement a specific type of event. For example, we have an ArrivalEvent object that is created when a customer enters the restaurant.
- Priority queue: A priority queue where all events will be enqueued upon creation. The queue will be sorted with respect to time.
- Waiting queues: Since it is possible that a customer might have to wait for a cashier or an order might have to wait to be assembled by a cook or to be cooked by a grill.
- The various Employee objects that the restaurant will contain.
- Lists to store the various employees.
- A Restaurant object: whose primary responsibility will be dequeue events from the priority queue and ask them to perform their actions.

Implementing the Restaurant:

The restaurant will be the central object within our system. It will create and contain all other objects within RestaurantSim. Its primary operation will be to dequeue an event from a priority queue, shared by all events, and ask the event to perform its action. It will repeat this procedure until the time of an event is equal to the restaurant's stop time. The restaurant will then print out the statistical information collected during its operation. Below is a high-level pseudocode of the Restaurant object.

```
Restaurant(String inFile, String outFile,
           long stopTime){
      // creates the employees and the inventory
      ReadFileAndSetupRestaurant(inFile);
      // now start operation
      startOperation();
}

void startOperation(){
do {
    if(!priorityQ.isEmpty()){
      Event e = priorityQ.dequeue();
      e.performEvent();
            }
}while(stopTime >= e.getTimeOfEvent());
// Compute the statistical information and write to
// outFile
computeStatisticalInformation(outFile);
```

The *ReadAndSetupRestaurant()* function creates the employees and the inventory as specified by the inputFile. Thus, the Restaurant contains lists of employees and the inventory within the Restaurant:

```
// list of employees
ArrayList cashierList, cookList;
//list of inventory
ArrayList mealsList, grillList.
```

The employees and inventory of the restaurant will be used by many events. For example, both the ServeCustomerEvent and the CashierFreeEvent will require a cashier to carry out their actions. For this reason, the employee and inventory lists within the restaurant will be globally visible to all objects within RestaurantSim.

RestaurantSim will also need waiting queues in case a customer has to wait for a cashier or an order has to wait to be cooked or assembled. Implementation details of these waiting queues are described below.

Implementing queues

RestaurantSim contains the following queues:

- A priority queue, called mainEventQ, that contains events that are prioritized by their time of occurrence
- Several waiting queues that are used when an object needs to wait for service.

The priority queue is an instance of class PriorityEventQueue. PriorityEventQueue extends EventQueue and implements the *insert()* function that inserts an event in the queue with respect to its time. In particular, it will insert an event in front of another event in the list that has time greater then or equal the one that is being inserted. All events get enqueued in this queue and are dequeued from this queue by the restaurant. This queue is therefore shared among all events.

The waiting queues are instances of **class** EventQueue. An EventQueue is a queue of Events and provides the basic functionality of a queue. RestaurantSim contains the following waiting queues:

- cashierQueue, which is used when a customer has to wait for a cashier
- cookQueue, which is used when an order has to wait to be assembled by a cook and,
- grillQueue, which is used when an order has to wait to be cooked using a grill.

Notice that the waiting queues are of type EventQueue. This means that when an entity (cashier, cook, grill) has to wait to be serviced, we enqueue the last event that occurred before the wait. For example, if a customer needs to wait for a cashier, then we

enqueue its ArrivalEvent in the waiting queue. A question that raises here is why do we enqueue the entire event instead of just the entity waiting for the service. The reason for this is that we wish to measure the time the entity had to wait for the next service. Thus, we store the previous event performed since it will be used to compute this information.

Implementing the Events

All events within RestaurantSim have the following in common:

- A time specifying when the event is to occur
- The action to be performed upon occurrence

Thus, one way to implement events is to have an abstract class Event, which will contain the characteristics common to all events. In particular, all events will have the following functions in common:

```
void setTimeOfEvent(long time);
long getTimeOfEvent();
void performEvent();
```

The function *performEvent*() performs the actual action associated with an event. For example, ArrivalEvent's *performEvent*() function checks to see if there is a cashier available to serve the customer. If so, it generates a ServeCustomerEvent. If not, it enqueues itself in the cashierQueue. Since each event will have its own implementation of the *performEvent*() function, the function will be declared as **abstract** in our **class** Event. All other events will then **extend** the **class** Event and provide their own definition of the *performEvent*() function.

Implementing the Employees

Our implementation of the employees within RestaurantSim will follow a structure similar to that of events. We have a class Employee which contains the functions and fields common to all employees. In particular, it contains the following functions:

```
Employee(int Id, long startTime, double salary)
// returns the employee id
int getNumber();
// employee stops working at this time
void setStopTime();
// sets the employee to busy or free
void putBusy(boolean status);
// returns true if the employee is busy
boolean isBusy();
//sets the empoyee to fired or employed
void putFired(boolean status);
// returns true if the employee is fired
boolean isFired();
// set the employees new salary
```

```
void setSalary(double newSalary);
// returns the employees current salary
double getSalary();
// returns the employees start time
long getStartTime();
// return the time the employee stopped working
long getEndTime();
// starts an employees idle time, called everytime an
// employees status is set to free
void startIdleTime();
// stops an employee idle time, called everytime an //
// employees status is set to busy
void stopIdleTime();
// returns the employee total idle time
long getTotalIdleTime();
// returns the employee's total salary
double getTotalSalary();
 // the average utilization of this employee
float getAverageUtilization();
```

Each **class** of employee then extends the Employee **class** and add the additional functions that it requires. For example, the **class** Cashier extends **class** Employee and adds the function, *getSales*() which returns a cashier's total sales.

This concludes our discussion of the implementation of RestaurantSim in Java.

Implementation in Haskell

Let us now take a look at the implementation of RestaurantSim in the modern functional language Haskell.

Following the principles of good software design, the implementation of ResturantSim in Haskell is broken up into the following modules:

- The Restaurant module
- The EmployeeInterface module
- The EventInterface module
- The AllEmployees module
- The AllEvents module
- The EventQueue module

Implementing the Restaurant module:

The Restaurant module is the central module within RestaurantSim. It has the following responsibilities:

- To create, setup and start a restaurant's operation
- To print out statistical information collected during the restaurant's operation.

The Restaurant module also implements the actual Restaurant. The Restaurant is implemented as a data type and contains the following fields:

- stopTime: which specifies the time when the Restaurant will stop its operation
- mainEventQ: where all events get enqueued upon creation.
- employeeList: which contains all the employees that work at RestaurantSim
- waitingQueue: which contains all the events that waiting are for service.
- StatInfo: which is of type StatisticalInformation. StatisticalInformation is a data type that contains all the statistical information that RestaurantSim will compute.

Calling the *newRestaurant* function and passing it the appropriate parameters accomplishes the task of creating, setting up and starting a restaurant's operation. The *newRestaurant* function is implemented as follows:

```
newRestaurant stopTime inFile =
      startOperation (setupRestaurant stopTime inFile
                      createRestaurant)
```

The *createRestaurant* function returns as its output a new Restaurant with all its fields initialized to either null or 0. The *setupRestaurant* function takes a Restaurant as one of its inputs and updates it by initializing the Restaurants' fields to the values specified by inFile. It then returns this updated Restaurant as its output.

This updated restaurant can now start its operation. This is accomplished by calling the *startOperation* function. The *startOperation* is a recursive function that takes a Restaurant as its input and returns an updated Restaurant that has finished its operation as its output. It is implemented as follows:

```
startOperation rest
     |  (stopTime rest) >= getTimeOfEvent event  = rest
     | otherwise = startOperation (performEvent event newR)
         where
       (event, newQ) = dequeue (mainEventQ rest)
    newR = { mainEventQ = newQ }
```

The Restaurant module also contains a *computeStat* function, which computes the statistical information collected during the Restaurants' operation, and output an updated Restaurant that reflects these changes. This function can now be called giving it the updated Restaurant, produced by *startOperation*, as one of its inputs. The *computeStat* function takes the name of file, where the output is to be printed, as its other input.

It should now be clear that the RestaurantSim program is run as follows:

```
ComputeStat (newRestaurant stopTime inFile) outFile
```

Implementing the EmployeeInterface:

The EmployeeInterface module is the module that is shared by all employees. It defines the **data** type Employee that is used to construct the different types of employees.

The data type Employee is implemented as follows:

```
data Employee = Cashier { inCommon:: CommonOfEmployees,
                          totalSales :: Double, … }
              | Cook { inCommon :: CommonOfEmployees, … }
              | …
```

CommonOfEmployees is a **data** type that consists of the fields to all employees.

Implementing the AllEmployees module

The AllEmployees module defines the functions common to all employees. These functions are similar to the ones defined in our implementation of RestaurantSim in Java with one exception. In our Haskell implementation we do not associate with each employee a variable to specify whether or not the employee is busy. The reason for this is that the employeeList contains only the employees that are free. Thus, when the service of an employee is required, it is removed from the employeeList. Upon the completion of its service, at which point it is considered free, it is added back to the employeeList.

A question that one might ask at this point is the following: If employeeList contains all the employees, regardless of their type, then how does one retrieve a specific type of employee from the list? A specific type of employee is retrieved from the employeeList through the use of the higher-order function **getOne**. In particular, a specific type of employee is retrieved as follows:

```
(theEmply, newList) = getEmployee emplyType employeeList

getEmployee ofType xs = getOne (checkType ofType) xs

getOne :: (a->Bool)->[a]->(a, [a])
getOne p (x:xs)
  | p x       = (x, xs)
  | otherwise = (r, x:ys)
 where
  (r, ys) = getOne p xs
```

where, the *checkType* function compares two employees to see if they are the same type and returns true if they are, false otherwise. The function *getOne* is then used

to retrieve a specific type of employee. Note that the function *getOne* assumes that the list is not empty.

The AllEmployee module also contains functions that are specific to an employee type. For example, it defines the *getTotalSales* function that is used to get the total sales of a cashier.

Implementing the EventInterface module

The EventInterface module is the module that is shared by all events. It defines the **data** type Event and is used to construct the different types of events.

The data type Event is implemented as follows:

```
data Event = Customer Arrival { time :: Time,
                                cust :: Customer }
           | Serve Customer { time :: Time, … }
           | …
```

where, *time* specifies the time the event is to occur.

Implementing the AllEvents module

The AllEvents module defines the functions common to all events. In particular, it defines the *getTimeOfEvent, setTimeOfEvent, and performEvent* functions. Below is a description of the functions:

```
-- takes an event and returns  its time of
 -- occurrence
 getTimeOfEvent :: a -> Time
 -- update an events time to the time given as
 -- input
 setTimeOfEvent :: Time -> a -> a
 --b is the entity on which the event will perform
 -- its action
 performEvent :: a -> b -> b
```

The *performEvent* function takes as one of its inputs, an entity upon which to perform the action of the event. In our case, this entity is Restaurant. The *performEvent* will take a Restaurant as one of its inputs, and output an updated Restaurant that results from performing the action of the event.

The *performEvent* function is redefined for each event, since each event will perform a different action depending on its type. For example, the action associated with an Arrival Event differs from the action associated with a Customer Leaves Event. Thus, both the events will need to provide their own definitions of the *performEvent* function. In particular, we have

```
performEvent (Evnt t (Arrv Cust)) =
```

```
                --its defintion

        performEvent (Evnt t (CustLeaves t)) =
                -- its definition
        …
```

Implementing the EventQueue module:

The EventQueue module consists of an implementation of a priority queue. A list is used to model the queue. Thus, we have the following data type:

```
data EventQueue a = Qu [a]
```

Functions are then defined and exported from the **module** EventQueue that allow Qu [a] to be treated as a priority queue.

Comparing the two implementations

The two implementations will be compared with respect to the following criteria:

- Language attributes
- Code size

The two implementations were very similar except for a few differences. I will first discuss the advantages that the Java implementation had over the Haskell implementation and the features of Java that led to these advantages.

Inheritance in Java

Inheritance in Java provides the means through which a software application can be extended to support additional functionality. It also enables new applications to reuse code implemented my existing applications.

Using inheritance, RestaurantSim can be extended in several ways. Below we look at a few of them.

Adding additional types of employees:

Using Java's feature of inheritance, the design of RestaurantSim can be extended to contain new employees, without requiring any of the existing classes to change. We simply create a new **class**, which extends the existing **class** Employee. For example, we can add Janitor's to RestaurantSim as follows:

1. Create a new **class** Janitor that extends **class** Employee.
2. Add to **class** Janitor the additional functionality that it needs.

In Haskell, adding new types of employees requires a modification to the data type Employee in the EmployeeInterface. In particular, Employee is modified to contain Janitor as one of its constructors:

```
data Employee = Cashier { inCommon :: CommonOfEmployees,
                          totalSales :: Double, …}
              | …
              | Janitor { inCoommon :: CommonOfEmployees,
                          --additional fields }
```

If the janitor provides additional functionality then these functions would need to be implemented either in a new file or the existing AllEmployees module.

One approach that would prevent existing files from being modified is to create a **type class** Employee and then make the different types of employees **instances** of the class. However, this will introduce a lot more work. Since there are several functions that all employees have in common, making each type of employee and **instance** of **class** Employee requires that these functions be defined again for every new employee type. To avoid this extra work, Employee was implemented as a data type.

Adding additional events:

Similarly, the design of RestaurantSim can be extended to contain additional events. Additional events are implemented by extending the existing **class** Event. For instance, we can add a CleanStore event to RestaurantSim as follows:

1. Create a new **class** CleanStoreEvent by extending the **class** Event.
2. Define the *performEvent*() function to perform the actions specific to this event.

Note that no existing classes need to be modified just by adding a new type of event.

In Haskell, just like employees, adding new types of events would require a modification to the **data** type Event in EventInterface. In particular, Event will now contain the additional constructor CleanStore:

```
data Action = Arrv { time :: Time, … }|
            …
            CleanStore { time :: Time, … }
```

A modification to AllEvents module will also be required, by adding a definition of the *performEvent* function for the CleanStore event.

Update operations and static types:

Because Java allows static variables, we were able to organize the computation of statistical information collected during the operation of RestaurantSim. In particular, each object was responsible for computing the statistical information relevant to it. For example, the **class** Cashier was responsible for computing the average wait time for a cashier and its total sales.

Because Haskell does not support static variables, we were unable to organize the computation of statistical information in the manner described above. Instead, we were forced to save all the statistical information in one place, namely the data type Restaurant. This increased coupling between the data type Restaurant and the other entities within RestaurantSim.

An approach that would reduce coupling and prevent all the statistical information from being saved within that data type Restaurant, is to have each object keep track of its own statistical information. For instance, each cashier will compute its average wait time. At the end of its operation, Restaurant will then compute the average wait time of a cashier by summing the waiting times for each cashier and then dividing it by the number of cashiers. This approach was not taken simply because it was overlooked.

Advantages of Haskell

We will now take a look at the advantages that the Haskell implementation had over the corresponding Java implementation.

Polymorphism

One of the key features of functional languages is polymorphism. This feature allows programs to contain generic functions that can be applied over many different types. For example, polymorphism in Haskell allows us to use generic list manipulating functions over any type of list.

Creating the list of employees:

In Java, there is no way to maintain type-information when storing objects in containers, except through the explicit creation of a type-specific container. When stored in a container, all objects get converted to the type Object. Hence there is no way to distinguish between the types of two objects stored within the list.

This led to create separate lists for each type of employees and waiting queues. For instance, RestaurantSim contains a cashierList that is used to store cashiers. Similarly, it contains a cashierQueue that is used to store ArrivalEvents.

However in the Haskell implementation, only a single list of employees, [Employee], was needed, which contained all the employees. The *filter* function was then used to retrieve a specific type of employee. Similarly, we maintained a single EventQueue for all events waiting for service. A three-line function *filterQ* was then implemented to retrieve a specific type of event.

This simplification led to the creation of events that could be implemented on all employees, instead of having a separate event of each of type of employee. For instance, in Java we had the following events that are very similar except for the fact that they operated on different employees:

- FreeCashierEvent
- FreeCookEvent
- FreeGrillEvent
- HireCashierEvent
- HireCookEvent
- HireGrillEvent
- FireCashierEvent
- FireGrillEvent
- FireCookEvent

In Haskell, because all employees were stored in one list, namely employeeList, we were able to combine the above nine events to the following three events:

- FreeEmployeeEvent
- HireEmployeeEvent
- FireEmployeeEvent

Higher-order functions:

Another important feature that Haskell provides is the ability to define **higher-order** functions. A function is **higher-order** if it takes a function as an argument or returns a function as a result, or both. Using this feature, functions can be composed of parts – a general higher order function and some particular specializing functions.

The use of the **higher-order** *getOne* function:

In the Haskell implementation, we defined a higher-order function *getOne* that is used to retrieve a specific type of employee. The use of this higher-order function played an important role in combining the events mentioned above.

The use of polymorphism and higher-order functions leads to another simplification. When new types of employees are added to RestaurantSim, the new employee also needs to be added to the Restaurant's employeeList. In

Haskell, because there is only a single list of employees, no modification to the Restaurant module is required. However, in the Java implementation there are separate lists for each employee. Thus, a new list needs to be created for the new employee in **class** Restaurant.

The Java implementation can be modified so that only a single list of employees would be required. Futhermore, we would also have the ability to add and remove new employee types during run time. This can be accomplished by adopting two design patterns – Abstract Factory and Prototype.[1] The purpose of these two patterns is allow an application to be independent of how its products are created. In particular, we would have an AbstractEmployee **class** that will be responsible for creating different types of employees. For more information on what these design patterns are and how they are used, please refer to the …

Note that the two design patterns can be adopted in both the Java and the Haskell implementation.

Code-size

The code sizes for the two implementations were within a close range of each other.

- Code-size for the Java implementation : 1567
- Code-size for the Haskell implementation : 1004

Both the numbers represent the total numbers of lines of code.

Conclusion

Because the implementation of RestaurantSim in Java and Haskell were so similar, one conclusion that can be drawn is that a well-organized design leads to an easy to code and flexible implementation, provided that the language supports the features required by the design. In the case of RestaurantSim, we followed the design used in implementing simulation systems. The overall functionality of the Restaurant was simple:

- Dequeue an event and perform its action.
- The action in turn creates and enqueues other events.

Thus because both the implementations followed a similar structure their code size and implementation were very similar.

[1] For more information on what these design patterns are and how they are used, please refer to *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, Vlissides.

APPENDICES

CODE FOR THE IMPLEMENTATION IN JAVA:

Can be accessed from: www.cs.utexas.edu/users/noorani/cs379/java_code

CODE FOR THE IMPLEMENTATION IN HASKELL:

Can be accessed from: www.cs.utexas.edu/users/noorani/cs379/haskell_code

WORKS CITED

Thompson, Simon. (1999). *Haskell: the Craft of Functinal Programming*. Harlow, England: Addision Wesley Longman Limited.

Hughes, John. (1984). *Why Functional Programming Matters.* Sweden.

Maisel, H. Gnugnoli, G. (1972). *Simulation of Discrete Stochastic Systems.* Chicago, US: Science Research Association, Inc.

Gamma, E. Helm, R. Johnson, R. Vlissides, J. (1995). *Design Patterns: Elements of Resuable Object-Oriented Software.* Boston, US: Addision Wesley.