HaskellQuest: a game for teaching functional programming in Haskell

Rongxiao Fu

Master of Science

School of Informatics
University of Edinburgh

2018

Abstract

In this project, the prototype of a cross-platform serious text adventure game named HaskellQuest teaching functional programming in Haskell is implemented.

With a background story happening in another world, the player explores different Haskell concepts and solves corresponding puzzles in HaskellQuest. During this process, the player's choices at branching points will affect the development of the story; the player will edit his/her answer to each puzzle in a built-in code editor where only text in editable ranges can be changed; and the player's answer will be evaluated in an environment connecting to GHC. Besides, just like in a Haskell book, every concept in HaskellQuest comes with examples and detailed explanation.

This game sequentially covers the concepts of values, expressions, types, variables, lambda abstractions, function definitions, parametric polymorphism, lists, list comprehensions and recursive definitions.

A user evaluation was carried out at the final stage of this project. According to the user evaluation, most players gave positive feedback and agreed that they have learned new knowledge about Haskell by playing HaskellQuest.

Acknowledgements

I would like to express my deep gratitude to my supervisor, Professor Don Sannella, who always offers me useful guidance and instant help during the whole process of this project. I also would like to thank my friends who helped me test my software. At last, I would like to thank all the test users involved in the final stage of this project for their helpful feedback.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Rongxiao Fu*)

Table of Contents

Chapter 1

Introduction

1.1 Motivation and objectives

This section is based on my IPP [54].

Video game is a modern form of entertainment. *"In the fantasy worlds built by video games, the players can escape from real lives, explore the unknowns, challenge themselves, cooperate with others and release their pressure.[74]"* [54] From another perspective, each game is a complex creation supported by rapidly developing hardware and software industries. With respect to the software part, games, especially 3D games, are usually the places where the latest software technologies are applied.

"Programming is another child of this remarkable digital era and has already infiltrated into every corner of our daily life. Our computers, phones, cars, TVs and even toothbrushes are controlled by programs running silently on the chips. Programmer has also become the most demanding working position in many countries." [54] In this context, many people begin to learn programming and try to use this skill in their lives.

However, learning is not always an easy and happy process. The construction of a complete knowledge structure usually means repetitive practice and countless trial-and-errors. As for learning programming, writing a lot of code is an inevitable process. If a learner feels tired and starts playing games for a break, he/she is actually very close to an alternative to the traditional learning process. What a game can bring about does not have to be purely entertainment, and there could be educational purposes hiding beneath the surface of a relaxing or exciting game. *"At this moment, embedding the process of learning and practising into games sounds like a reasonable idea."* [54] With the attractiveness of learning while still having fun, the engagement of learning might also be improved.

While talking about learning programming, the first language is always an interesting topic. There are many so-called mainstream programming languages, such as Python, Java and C++. These languages have large communities and numerous clients. Many universities also use them as the languages for the introductory courses of programming. On the other hand, there are some not so popular but still attractive languages, and Haskell is one of them.

Haskell is a statically typed, purely functional programming language [58]. More information about Haskell will be introduced in Section 2.3. Many concepts in Haskell are closely related with theoretical computer science and mathematics, such as purity, laziness, functors, monoids and monads [55]. And sometimes the beginners may be frightened by these terms. It will be great if there could be a game to demystify these concepts and demonstrate them in an intuitive way.

And this is the goal of this project: designing and implementing a game teaching functional programming in Haskell. Affected by another game called ScalaQuest [33], this game is named HaskellQuest. Games for educational purpose like HaskellQuest are called serious games, and this concept will be explained in Section 2.1.

To start the development of this game, the genre of HaskellQuest should be decided. There are lots of game genres, such as action, role playing, adventure and strategy, with finer classifications under each large category. With respect to HaskellQuest, it is a game teaching programming, and although coding is not the only way of programming, it must be the most common way. If the players are going to write code in HaskellQuest, a text-based game can give the players a consistent visual experience. It will also be easier and more natural to explain some concepts in detail with text in a text-based game. Thus, as a typical kind of text-based game, text adventure is chosen as the genre of HaskellQuest. More information about text adventure games is introduced in Section 2.2.

Generally, in HaskellQuest, the player will learn a series of basic Haskell concepts and practise by solving puzzles. There will be a background story and NPCs (non-player characters) guiding the players, and there will be detailed description for each puzzle. This game should help the player build a basic knowledge structure of Haskell. In other words, HaskellQuest is designed to be a playable Haskell book.

According to the minimal implementation proposed in my IPP [54], the Haskell concepts covered by HaskellQuest should at least include expressions, function declarations, types, lists, list comprehensions and recursive definitions.

1.2 Achievements

Generally, the prototype of a cross-platform serious text adventure game teaching functional programming in Haskell is implemented in this project. Besides, the following achievements are accomplished in this project.

- A messaging interface displaying text messages and clickable options and a puzzle-solving interface involving stylised puzzle description and code editor are implemented. They both work smoothly and stably.
- A story data format encoding the development and branching of a story is designed and used to store the background story of HaskellQuest.
- A code editor where users can only edit the text inside editable ranges is implemented based on Ace Editor. A quotation notation is provided for the developer to define arbitrary editable ranges.
- A Haskell code evaluation environment is implemented. Both multiple answers to the same question and strict code comparison are supported.
- A background story is written to combine text adventure and Haskell education, and it is presented in the messaging interface.
- 24 Haskell puzzles with detailed descriptions are designed.
- The game sequentially covers the concepts of values, expressions, types, variables, lambda abstractions, function definitions, parametric polymorphism, lists, lists comprehensions and recursive definitions.
- A user evaluation was carried out at the final stage of this project. According to the user evaluation, most players gave positive feedback and agreed that they have learned new knowledge about Haskell by playing HaskellQuest.

1.3 Thesis structure

The remainder of this thesis is divided into four chapters.

- Chapter 2 gives the definitions and examples of serious game and text adventure game. ScalaQuest is analysed as an example.

- Chapter 3 introduces the design and implementation of each component in HaskellQuest. The background story and the knowledge structure of HaskellQuest are also introduced in this chapter.
- Chapter 4 is the results and analysis of the self-evaluation and user evaluation of this project.
- Chapter 5 concludes the thesis and proposes some directions for future work.

Chapter 2

Background

2.1 Serious games

A serious game can be defined as a game designed for other (usually educational) purposes besides entertainment [60, 81, 54]. According to my IPP [54], *"serious games can be applied in different areas [81], typical application include school education [53], military and medical training. Besides, serious games have already shown their ability to improve the engagement and effectiveness of learning in some scenarios [81]"*.

A recent example of a serious game is "Variant: Limits" developed by Triseum [46], which is an award-winning 3D adventure game teaching calculus to its players. In this game, the player will assist a character named Equa to save a dooming world ruled by calculus principles. In this process, the player will solve visualised puzzles and gradually learn calculus concepts such as limit and continuity. In this game, abstract mathematical concepts are intuitively visualised and converted into interactive challenges to be better understood. Aiming at the educational market, Variant: Limits also provides tools for instructors to track students' gaming and learning progress. According to the post-game survey report of Variant: Limits [47], 70% of the survey responses indicate that this game is both educational and entertaining.

Another example of a serious game, also an example of a game allowing players to practise programming skills is "Screeps" [35]. It is a MMO (massively multiplayer online) RTS (real-time strategy) sandbox game. The player needs to write JavaScript code to control his/her own colony in a world shared with other players, and try to collect more resources and conquer more territory. An important feature of this game is that the sandbox world simulation goes on even when the player is offline. According

to the players' review [36, 40], Screeps may not be suitable for completely beginners, but with sufficient background knowledge in JavaScript, the players' programming skills can be trained and improved by playing this game.

As for serious games that actually teach programming to beginners, "CodeCombat" [10] could be an example. Instead of solely being a game, CodeCombat is more likely to be a educational platform containing a large collection of game levels where the players need to write actual code to complete tasks. The languages supported by CodeCombat include Python, JavaScript and CoffeeScript. Since education is the main application of CodeCombat, some of its features are specially designed to be used in classrooms.

2.2 Text adventure games

Text adventure game, sometimes also known as interactive fiction [61], is a category of games where an adventure story is progressively presented in a text-based form and the player is involved in the construction of this story. In a text adventure, the player's commands or choices will affect the development of the story, and there are usually multiple endings for the player to explore. The traditional way for the player to influence the game progress is by typing a sentence, then the game engine will analyse it and give a corresponding response, but nowadays selecting from multiple options is also a common choice.

Figure 2.1 is a screenshot of Lifeline: Whiteout [25] on an Android phone. This game is one of the Lifeline text adventure series developed by Three Minute Games [24].

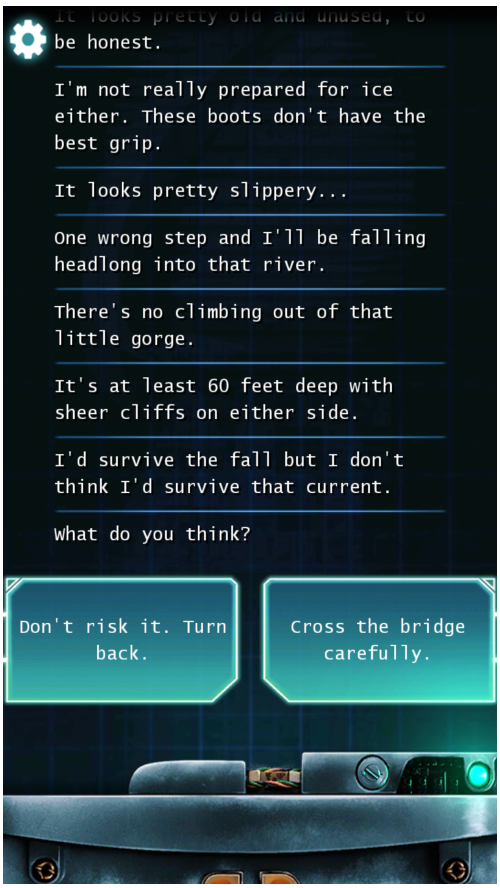

Figure 2.1: A screenshot of Lifeline: Whiteout

The story in Lifeline: Whiteout starts with a NPC named Adams waking up and finding himself lying on a frozen lake with a wrecked snowmobile nearby. Adams has lost part of his memory, but still manages to get in touch with the player using a portable communication device. The player's task is to help Adams get out of this dangerous situation and find his true identity.

An important design in Lifeline: Whiteout is that, delay is treated as one of its game elements. This feature is called "real-time text adventure". If Adams is talking with the player while doing something else, the messages from Adams are displayed on the screen with noticeable and reasonable delays as there would be in real life. And the player must wait when Adams is busy or sleeping. These intentionally added delays make the game more realistic. The player may imagine that this story is actually happening somewhere in this world and there is a poor guy named Adams waiting for help. Some abnormally long delays even make the player worry about Adams and frequently check the game status.

Another interesting design aspect of Lifeline: Whiteout is that, although the player

can give Adams instructions, Adams will not always follow them. For example, if the player tells Adams to investigate a faraway place, Adams may refuse or doubt because he thinks it is too dangerous. This feature can give the player the impression that Adams is a true character with his own thoughts.

The design of Lifeline: Whiteout, especially the concept of real-time text adventure, affected the design of HaskellQuest.

2.3 The Haskell programming language

According to the Haskell 2010 language report [58], Haskell is a general purpose, statically typed, purely functional programming language with non-strict semantics. The most famous and widely-used Haskell implementation is the Glasgow Haskell Compiler (GHC) [59].

The term "functional programming" indicates that a function is a first-class citizen in Haskell and compared with other programming languages, Haskell emphasises the composition of functions and the use of higher order functions.

Haskell is a pure language. Purity means that side effects, such as internal state, file input/output and variable mutation, are strictly constrained in Haskell. With purity, the expressions in Haskell have the property of referential transparency [77], which means that an expression is completely interchangeable with the values it evaluates to. Referential transparency makes it easier to reason about a Haskell program [55]. But sometimes, side effects, or the simulations of side effects, are expected in a practical program. In this case, Haskell uses monads to encapsulate side effects and make sure that they are all under control [55].

Haskell is a statically strongly typed language, which means that type-checking happens at compile time and no implicit type conversion is allowed. Besides, Haskell provides parametric polymorphism and ad hoc polymorphism (type classes) [55, 66], allowing more abstract and flexible definitions. Thanks to type inference, with all the benefits (error detection at compile-time, high level abstractions, etc.) of such a type system, there is even no need to write type signatures in most cases. As a rapidly developing type-system laboratory [55], the type system of Haskell is too powerful to be covered with a brief introduction here.

Haskell has non-strict semantics which is implemented with lazy evaluation in GHC [55]. Lazy evaluation means that in Haskell, the evaluation of an expression will be delayed until its result is needed. Lazy evaluation can help avoid unnecessary

computations and allow the existence of some useful constructions such as infinite lists, but it can also lead to performance issues and cause space leaks [55]. Thus, Haskell provides programmers with various tools (`seq`, lazy patterns, etc.) to control laziness on a finer scale.

2.4 ScalaQuest

ScalaQuest [33] is an online game teaching Scala programming and as its name suggests, this "HaskellQuest" project was inspired by ScalaQuest in some ways. The Kickstarter page with a detailed introduction to ScalaQuest can be found at [34]. ScalaQuest is still in its development and the content of this section is based on a beta version of ScalaQuest in March, 2018.

Scala is a statically typed multi-paradigm programming language [62] where functional and object-oriented programming are combined. It uses the JVM (Java Virtual Machine) as one of its runtimes and has seamless compatibility with Java. The language design of Scala is influenced by Java and Haskell (and other languages) [62]. As an example of a Scala program, the following code is a function which reverses a list.

```
def reverse[A](x: List[A]): List[A] = {
        @tailrec
        def go[A](s: List[A], acc:List[A]): List[A] = s match {
                case Nil => acc
                case y::ys => go(ys, y::acc)
        }
        go(x, Nil)
}
```

Noteworthy languages features shown by the code above include parametric polymorphism, pattern matching, nested function definition and tail recursion annotation.

As for the game ScalaQuest, Figure 2.2 is a screenshot taken at Level 1 of ScalaQuest.

Figure 2.2: A screenshot of ScalaQuest showing a question in Level 1

In this level, the player's task is collecting all seven gems on the road by answering a series of questions about the basic concepts in Scala, such as values, variables and types. Putting the questions aside, as a game, ScalaQuest has decent user interface with interesting details and reasonable interaction logic. It also has a background story about the career of a wizard's apprentice in a world called `DataLand` [34], so the player will not be confused by the art style of environment and characters. And in the coding interface, the inline code and descriptive text are clearly distinguished with different colours, improving the readability.

A significant feature of ScalaQuest is that there are lots of questions and the same concept will be covered multiple times. This is also mentioned by the authors of ScalaQuest on the Kickstarter page [34]. As a direct way of enhancing the memory, repetition is an important element in both gaming and learning, but it is not easy to locate the boundary between meaningful and tedious repetitions. ScalaQuest avoids making the player bored by examining the same concept from different aspects, and this normally works.

However, some problems occurring in ScalaQuest cannot be ignored.

The first problem to be discussed is that most questions in ScalaQuest are presented to the player directly without adequate introductions to the involved concepts. Although the player can use the blue "Hints" button to get several hints about the current question, the fragmented information provided by hints is not as useful as a complete

and well-organised question description. According to the statement in my Informatics Project Proposal [54], *"ScalaQuest is more likely to be a collection of after-class exercises than lecture notes for Scala learners"*. This gaming/learning mode might be more suitable for those who have learned about Scala to practise their skills, but for Scala beginners, in my opinion, some guidance and explanation is necessary.

The evaluation process of the player's code in ScalaQuest is also not so satisfatory. Figure 2.2 has already shown an example. The question seems to be a relatively easy one, and even the hint is a straightforward description of the correct answer, that is, `val area = l * w`. However, the code in the screenshot, `val area = w * l`, is not accepted. There is no reason for this line of code to be wrong because the multiplication of real numbers is commutative. A reasonable inference is that, ScalaQuest only compares the parsing results of the correct answer and the player's code. In this case, unless specifically designed, every question has only one correct answer. This limits the possibility for exploring different solutions, and sometimes leads to confusion if multiple answers obviously exist.

Another example is shown by Figure 2.3. This is the last question the player needs to solve in level 1. By solving this question, the player can open the door to the next level. Being different from the questions for collecting gems, the Scala code in this question is actually evaluated. Two problems occurred in this question. The first one is the lack of a running time monitor, that is, if the player's code leads to an infinite loop, the game will become unresponsive. The other problem is a mistake in the question description. The description says *"Iterate values from zero to the number of gems you collected"* and *"for odd ones print "pull""*. There are seven gems on the road, so four `"pull"`s should be printed (for `1`, `3`, `5` and `7`, respectively). However, according to the evaluation feedback in the lower part of Figure 2.3, only three `"pull"`s are expected.

Figure 2.3: A screenshot of ScalaQuest showing the last question in Level 1

The last thing to be mentioned here is not only a problem of ScalaQuest, but also a common problem in educational serious games, that is, the game elements are not closely related with the concepts to be taught. Similar topic is also discussed in my IPP [54]. In ScalaQuest, the player collects gems in Level 1, collects the body parts of stone sculptures in Level 2 and fight the demons in Level 3. These actions are not related with Scala. In other words, Scala is not an essential part in these tasks. But the question in Figure 2.3 can be considered as a relatively good example of connecting Scala program with the game world, because the printed output is the sequence of actions required to open the door. Generally speaking, building reasonable and consistent connections between gaming and learning is not easy, and this problem is very likely to occur in HaskellQuest as well.

Chapter 3

Design and Implementation

The design and corresponding implementation of each component in HaskellQuest is introduced in this chapter.

- Section 3.1 gives the prototype design of HaskellQuest.
- Section 3.2 explains the choices of development tools.
- Section 3.3 introduces the design and implementation of the messaging interface.
- Section 3.4 introduces the design and implementation of the puzzle-solving interface. The built-in code editor and the evaluation environment is also introduced in this section.
- Section 3.5 gives a summary of the background story written for HaskellQuest.
- Section 3.6 shows the structure of HaskellQuest if it is treated as a playable Haskell book.

3.1 Prototype design

Presented by Figure 3.1 is a prototype of the UI (User Interface) of HaskellQuest designed at the start of this project. The final implementation is different from this design, but their basic structures are the same. Screenshots of the final UI are given in Chapter 4.

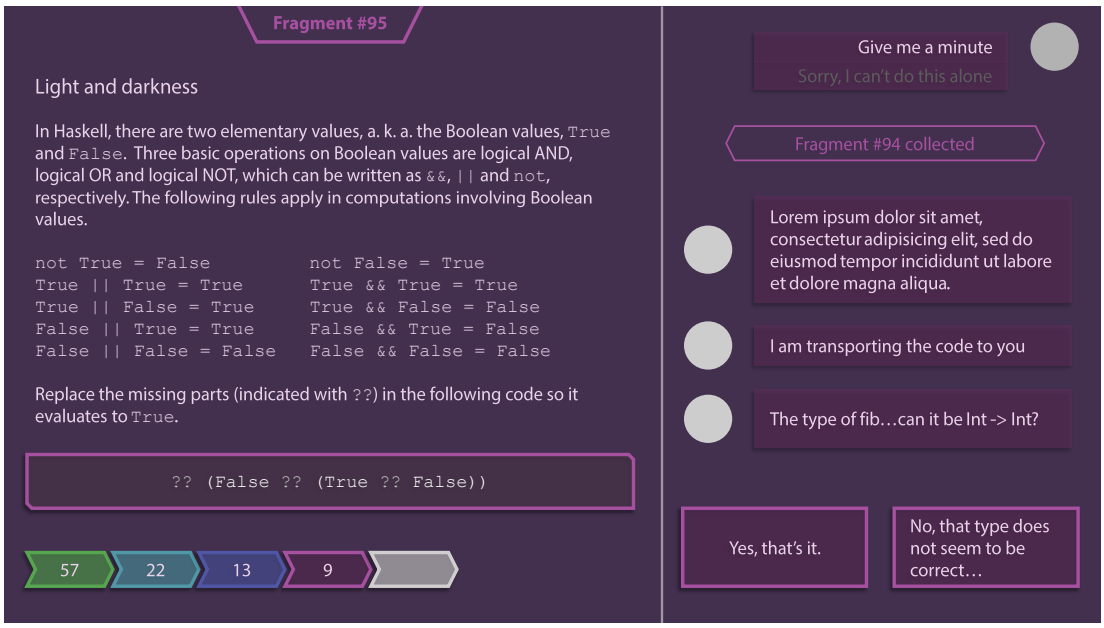

Figure 3.1: The prototype UI design of HaskellQuest. Lorem Ipsum [26] is used to provide some text.

The interface is divided into two parts by an white line. The right-hand-side part is a messaging interface where messages and clickable options are displayed. Messages sent by the NPCs are left-aligned in the message list, and the system messages are centred. The player can sent messages by clicking on the options popping up from the right-bottom, and the chosen one will be presented as a normal message with the other one being displayed in a dimmed colour. The left-hand-side part is a puzzle-solving interface where the player can read the question description and answer questions in a built-in code editor. The colourful blocks at the bottom-left corner with numbers on them are intended to be different types of rewards collected by the player, but in the final implementation there is only one type of reward.

The design of the messaging interface is affected by Lifeline [24] and Android Messages application [5]. The colour scheme of the whole interface is based on the official website of Haskell [18].

3.2 Development tools

As a famous and popular cross-platform game engine with a large community, Unity [41] was originally chosen as the development framework for HaskellQuest. Unity uses C# as its primary script language [42] and is capable of creating 2D and 3D

games with outstanding visual effects. A collection of games developed with Unity can be found at its official website [17]. Since HaskellQuest is a text-based game, the 2D game development workflow [44] in Unity should be enough for the presentation of text content and the interaction with players, and Unity provides the possibility of adding more interesting game elements if needed. However, as both a Unity and C# beginner, I spent time learning and getting used to this game engine, and Unity was found to be not the most suitable tool for this task.

Actually, at the beginning stage of this project, Unity showed its capability of creating UIs [43], and there is an official tutorial about making a text adventure game in Unity [21], although graphical UI development is not included in this tutorial. At that stage in the project, I was unfamiliar with Unity and was learning to use it, but to complete this project in time, a faster development speed was required. Thus, I was looking for a tool which can accelerate the implementation of a messaging interface, and after comparing several Unity assets, a UI presentation framework called MarkLight [27] [28] was chosen. Instead of directly manipulating the UI elements in Unity, with MarkLight, developers can describe UIs with an HTML-like markup language called XUML, and MarkLight adopts the MVVM (Model-View-ViewModel) pattern [78] to manage the processing and presenting of data. Exemplar code can be found at [27].

MarkLight simplifies the development of UIs in Unity, and it is powerful enough in most cases. However, after one week of trial-and-error, MarkLight showed its weakness. It does reduce the work required to create a formatted UI in Unity, but is not flexible enough sometimes. For example, to create a button in MarkLight is easy. The following code and Figure 3.2 is a MarkLight button with fixed width and multi-line text.

```
<Button
  Width = "300px"
  Text="Button Text\nSecond Line"
  FontSize = "40"
  AdjustToText = "Height"
  Click = "ClickHandler"
/>
```

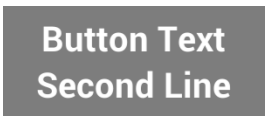

Figure 3.2: A normal example of MarkLight button

This button seems to work well and it does adjust its height to the text. However, if the text length exceeds the button width, for example, setting the `Text` field to `"Button Text Some Extra Text\nSecond Line"`, then as shown by Figure 3.3, instead of wrapping to a new line, the button text will overflow.

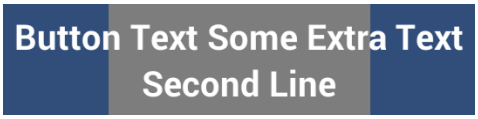

Figure 3.3: A MarkLight button with overflew text. The background is shown in blue.

This problem can be fixed by modifying the source files related with the definition of button in MarkLight. In the meantime, other similar or even more complex problems were discovered, such as list items with fixed heights and unsupported image insertion, and to fix them, even more source files of MarkLight would need to be modified. In order to avoid introducing new problems in this process, I would need to understand the MarkLight source code more thoroughly. This makes the use of a library meaningless and was the major cause of the framework change.

Electron [7] was later chosen and became the development framework currently used to build HaskellQuest. According to its official documentation [3], Electron is a cross-platform desktop application development framework in which Chromium [9] and Node.js [29] are packed together as supportive tools into a single runtime, so the user can use web development technologies in a desktop application. In other words, by using Electron, the UI of HaskellQuest can be implemented as a webpage using HTML, CSS (Cascading Style Sheets) and JavaScript. Besides, Electron provides the necessary interfaces required for HaskellQuest as an actual desktop application to communicate with other programs and the operating system. Compared with the Unity community, the web development community is even much larger, and has abundant tools and libraries allowing more customised design.

Although developing an application as a webpage has significant flexibility, implementing an UI directly with HTML and other basic tools is still a cumbersome and sometimes onerous task. A UI library called React [31] is used to reduce unnecessary

workload, and using React together with Electron is also a common combination.

React is an open source JavaScript frontend library developed by Facebook [15] for building UIs in an efficient and flexible way [31]. React is widely used in various websites [37], such as Facebook [14], Instagram [20], Airbnb [19] and Bitbucket [6].

A noteworthy and also interesting point about React is that the concept of functional programming and immutable data is applied in this library [79]. An UI created with React consists of building blocks called components. According to Pete Hunt in [79], React shows the idea of “referentially transparent UI", which means a React component can be treated as a black box taking some inputs and producing an output representing the content to be displayed, and the same inputs will always lead to the same output. A new output from a React component will be compared with its previous version to decide which part should be updated. This method releases React from keeping track of any changes in the underlying data, so it can focus on the content which is actually rendered. On the other hand, each React component can have a local state object, and a change of state will trigger a new updating process. React discourages in-place modifications on the state of a component. Instead, by explicitly calling the `setState` method, a new value of state can be set to replace the older one. The special part here is that state update in React can be asynchronous and batched [38], which means the state may not change instantly after a `setState` call and multiple `setState` calls can be merged.

Besides, thanks to the strong support for asynchronous programming in JavaScript [49], all the time-consuming operations, such as reading and writing files, or waiting for the response from Haskell compiler, are asynchronous, so the UI will not get stuck. And atomic file writing [48] is used in the whole project to guarantee the completeness of recorded data.

3.3 Messaging interface

HaskellQuest is designed to be a text adventure game, so the messaging interface becomes one of the most important components. In the fulfilment of basic functionality, the fictional character(s) emulated by the game and the game system itself will be able to send messages to the player, and the player’s choice between different options which may lead to divergent storylines, is again a message to the fictional character(s). Besides, to imitate the actual time required for a real person to compose a message, each NPC message is sent with an intentionally added delay.

To decouple the development of storylines and the implementation of the messaging interface, the story is decomposed and stored in local files, and the messaging interface is a renderer, which will load these story files dynamically, extract and organise required information, combine it with other graphical components and display them on the screen in a visually attractive way. In this case, project maintenance also becomes easier because the story can be modified without altering the source code of the messaging interface.

With respect to the current implementation, story is divided into chapters and each chapter is described by a JSON [51] file. This file format is chosen mainly because the messaging interface is implemented with React [31], which is a JavaScript library and JSON files can be easily loaded and edited with JavaScript or manually. Internally, a chapter is basically a line of title text and a list of sections, where each section is assigned a unique tag or identifier. Moreover, a section contains a message list, a question (which may also be referred to as a puzzle in the following text) and a branching instruction. Each message in the message list consists of three fields, its sender, content and a sending delay (in seconds). The question field contains a reference to the question folder which will be explained in Section 3.4. And as its name suggests, the branching instruction controls the developing of storylines; it can either be an unconditional jump to any existing section or a conditional branch which depends on the player's choice. Lastly, at the end of the message list, an extra delay field is added to control the delay of presenting the question or options.

A chapter with a single section jumping back to itself every 10 seconds can be described as follows. Besides those data fields mentioned above, for the purpose of evaluating the player's coding skills, each chapter carries an entry mark which must be achieved to enter this chapter.

```
chapterLoop = {
  title: "chapterLoop",
  entryMark: 0.0,
  sections: [
    {
      tag: "loop",
      messages: [
        {
          sender: "npc",
          delay: 10.0,
```

```
11            content: "message"
12          }
13        ],
14        endDelay: 0.0,
15        question: null,
16        branch: {
17          uncondTo: "loop",
18          options: []
19        }
20      }
21    ]
22  };
```

As for the rendering of the message list, its underlying idea is very straightforward. The appearance of a message on screen, such as its floating position and icon image, can be completely decided by its sender and content. Thus, for each message in a section, an object containing all the information required to rebuild this message without referring to the story file will be composed, and then inserted into the message list. This will cause a state change and React will update the screen content to display the new message. After the delay specified by the next message, the same process repeats. At the end of each section, a question (if available) will be presented, and then the branching instruction will be executed so the story can continue.

The player will not like it if he/she exits the game and must start over next time. Thus, to recover the game from where the player stopped, the game progress needs to be saved as file(s) frequently. Although the status of questions and answers is important and must be saved, it will be introduced in Section 3.4 and this section will focus on general progress information and the message list.

A method for keeping track of the progress information in this game would be to save the player's choices at conditional branches and reproducing the message list without delay. In other words, each reload of the game would actually replay the conversation part of the game at a high speed. Progress information generated in this way is clear and compact, but on the other hand, the regeneration of the message list introduces unwanted performance overhead and complexity, and the dynamic generation of messages, such as inserting system messages depending on the player's performance at arbitrary points, is not supported.

Thus, for HaskellQuest, the complete historical message list is saved as a single file, which can be directly loaded and rendered without extra computation. The game

progress is recovered by loading another file containing the location (chapter, section and index) of the current message and some other miscellaneous information. An example using the loop chapter is given below.

```
progress = {
  chptList: [
    {
      title: "chapterLoop",
      currentMsg: 0,
      currentSect: 0,
      entryMark: 0
    }
  ],
  currentChpt: 0,
  totalReward: 0,
  newChptIndex: -1
}
```

The whole story will consist of multiple chapters, so the game progress contains a list of visited chapters and the player can switch between them. Besides the necessary location data mentioned above, some extra information about each chapter, the total reward achieved and a field indicating if a new chapter is available are also included in the progress information.

3.4 Puzzle-solving interface

As mention in Section 3.3, the player is going to be presented with questions/puzzles at the end of some sections. This section is mainly about how this puzzle-solving interface is implemented.

3.4.1 Properties of a question

A typical question field in a story section looks like the following JSON object.

```
question: {
  tag: "q0",
  title: "Question Title",
  reward: 20.0,
  isOptional: false,
  isTimed: true,
  expectedTime: 300
}
```

The tag of a question is actually the name of a folder where the description and templates of the corresponding question are stored. This design detaches the details of questions from the story files so they can be managed separately. A question folder should at least contain three files, a Markdown file [13] for the question description, a template file for the editor introduced in Section 3.4.2, and a Haskell source file named `Main.hs` for the evaluation environment introduced in Section 3.4.3.

The title of a question is used to give a sensible name to the current question instead of using the tag directly.

The reward of a question is the highest mark the player can get from this question. When the player submits an incorrect answer, as a penalty, a fraction of the reward will be removed. The rule currently applied is for each incorrect solution, the current reward will be reduced to 80% of its previous value, until it is less than or equal to 50% of the highest reward.

A question can be either optional or mandatory. The player must answer a mandatory question correctly to continue the game.

A mandatory question can also be timed. A timer will show up if the player clicks into a timed question. If it takes more than the expected time (in seconds) for the player to complete the question, a fraction of the reward will be removed. The rule currently applied is described by the following pseudo-code.

```
removedReward = (
  0.5 * highestReward *
  max(timeTaken - expectedTime, 0) /
  expectedTime);
currentReward =
  max(currentReward - removedReward, 0.5 * highestReward);
```

This reward reduction mechanism will work together with the incorrect solution penalty, so the code above indicates that if the player can provide a correct solution

as his/her first submission, he/she will have twice the expected time to compose the answer before the reward is reduced to its lowest value. A special case is that if the player exits a timed question without answering it correctly, the reward will be directly reduced to its lowest value.

The purpose of such a reward system is to encourage the player to think carefully before submitting an answer, and a timed question requires the player to understand and solve the puzzle faster. Besides, the pressure from a timed question and the relief that an in-time correct submission may bring about can also contribute to the gaming experience.

3.4.2 Built-in editor

The player will be asked to edit his/her answer within the puzzle-solving interface and an editor with syntax highlighting can bring a much better experience than a plain text input area. Thus, a built-in code editor is required in HaskellQuest.

Two popular code editors implemented in JavaScript, Ace Editor [4] and CodeMirror [11], were compared, and Ace Editor was eventually chosen because its corresponding React component library [32] is easier to use and well-documented. Figure 3.4 shows a screenshot of Ace Editor with Haskell syntax highlighting.

```
mean :: Double
mean = (10.01 + 10.05 + 9.92 + 10.01 + 9.97) / 5
result :: Bool
result = mean >= 9.95 && mean <= 10.05
```

Figure 3.4: A screenshot of Ace Editor as a React component.

Although many fancy text editing functionalities are provided out-of-the-box by Ace Editor, simply embedding it into the puzzle-solving interface is not enough. For example, a question the player is going be asked at the beginning stage of this game is to replace the occurrences of `??` with Haskell code in the following expression so it evaluates to `True`.

`?? (False ?? (True ?? False))`

One of the straightforward solutions would be

`not (False && (True || False))`

And solutions such as the following one, although not the best, is still correct.

`let n = not in n (False && True && (True || False))`

However, if the player can freely edit other parts of the expression besides the occurrences of `??`, a simple `True` will be a correct answer and the question becomes meaningless. On the other hand, if multiple (infinitely many in many cases) solutions exist, the answers should not be limited to several “standard" ones. Thus, the size and number of editable text ranges should be restricted, in other words, there should be some read-only text ranges.

Setting a range of text to read-only is not natively supported in Ace Editor, and the only functionality supported is setting the whole editor to read-only mode. This feature of Ace Editor can be used for an intuitive implementation of read-only ranges, by tracking the position of the cursor and entering read-only mode if the cursor is inside an uneditable area, but this method is problematic and was abandoned because the cursor updating event is emitted very frequently, making it relatively hard to locate useful information.

The second method is tracking the modified text range and if it violates any of the read-only ranges, this step of modification will be recovered/cancelled. This method works well in most cases and was eventually applied in HaskellQuest.

One key element of this method is the implementation of the read-only range. Actually, there is no significant difference between implementing read-only and editable ranges, because if a range is not read-only, it must be editable. And in HaskellQuest, editable range is chosen and is implemented with anchors.

An anchor, as its name suggests, is an object attached to a relatively fixed position in a text stream and when the text is modified, the anchor will move together with it. For example, suppose `AB|CD` is a piece of editable text and the blue vertical line represents an anchor; then, if `123` is then inserted between `A` and `B`, the text and anchor now become `A123B|CD`.

A pair of anchors can define a range. In HaskellQuest, a quotation syntax is defined to denote the editable ranges conveniently. Concretely speaking, the text shown in the built-in editor is generated from a template (see Section 3.4.1) where text quoted by `[- -]` will be considered editable. The `[- -]` notation will also be used in the other chapters and sections to denote editable ranges. Multiple editable ranges in a single template are allowed, but for a clearer demonstration, only one editable range will be used in this section. For example, parsing the template `ABC[-123-]DEF` will generate `AB|C123D|EF` where the underlined range is editable and the anchors are shifted one character away to avoid the boundary problem. Specifically, if the anchors are inserted at exactly the boundaries between the editable and read-only ranges, that is,

ABC|123|DEF, and a character X is then inserted between C and 1, an ambiguous situation would occur – is this a valid insertion in the editable range (ABC|X123|DEF) or a violation to the read-only range (ABCX|123|DEF)? The result depends on the implementation of the editor's anchor moving algorithm. By shifting the anchors one character away from the boundaries, the ambiguity is solved. The example case now certainly becomes a valid insertion (AB|CX123D|EF). And if the character X is inserted between B and C, this insertion will be refused because it is in the read-only range anyway.

Once the anchors are all set up, we can move on to the detection of read-only violation and recovery. In Ace Editor, all the operations on text are simplified to the combinations of insertions and removals. So, by listening to the document change event of Ace Editor, the script could know the action type of the current change (`insert` or `remove`), the affected range and associated text content. For example, removing BC from ABCD will trigger the change `remove 1 3 'BC'`, and inserting 123 between B and C in ABCD will trigger the change `insert 2 5 '123'`. In the event handler, the script will check if the current change violates the read-only ranges, in other words, it will check if the affected range is within an editable range. Because the text is already changed and the locations of anchors are all updated before the script enters the event handler, so the affected range is actually compared with the previously recorded locations of anchors. For a removal action, both its start and end locations are checked. But for an insertion, only the start location needs to be checked, because if the insertion starts inside an editable range, it will push away the successive text, which makes the inserted text always stay inside that editable range.

Subsequently, if a change is valid, the recorded anchor locations will be synchronised with current anchors and no more operations will be performed on the text. Meanwhile, if a change violates the read-only ranges, it is recovered/cancelled with a change-reverting function provided by Ace Editor. The working principle of this function is conceptually easy, because `insert` and `remove` revert each other. A noteworthy point is that reverting of a change is still a change and will again trigger a document change event, so a condition checker needs to be added to prevent recursively calling the event handler.

However, change-reverting is not enough to completely recover the state of the current document because the anchors are already moved with the text. Thanks to the recorded anchor locations, all the anchors can be moved back to their previous locations after the text is recovered.

An issue of the current approach is that it does not work properly with Chinese input methods: Pinyin (the romanisation of Chinese) characters can be inserted into read-only ranges. A similar issue occurs for Japanese input methods, but not all input methods for languages using non-roman characters are affected. The reason is still unclear, but is probably related to the implementation of anchors. This issue can actually be solved by using a preserved document to recover text instead of change-reverting, but this workaround introduces some other minor problems such as confusing cursor positions, and requires further improvement.

Another issue found later is caused by the undo (`Ctrl+Z`) and redo (`Ctrl+Y`) operations. The following table shows an example of using an undo action to modify a read-only character. For a clearer demonstration, the process shown in the table is simplified and some details are left out.

Action	Text	Head of undo stack
None	`ABC123DEF`	`[]`
Insert	`A X BC123DEF`	`[insert 1 2 'X']`
Recover	`ABC123DEF`	`[remove 1 2 'X', insert 1 2 'X']`
Undo	`A X BC123DEF`	`[insert 1 2 'X']`
Recover	`ABC123DEF`	`[insert 1 2 'X']`
Undo cont.	`AC123DEF`	`[]`
Recover	`A X C123DEF`	`[]`

The editor maintains an undo stack of historical changes, and the head of the undo stack is a list of the most recent changes which will be reverted sequentially once an undo operation is triggered. During the execution of undo, changes will not be pushed into the stack (to avoid recursion), but the document change events are still handled. According to the table, when `insert 1 2 'X'` is reverted by undo, the change event is naturally its reversion `remove 1 2 'X'`, but the actually removed character is `B`. Thus, after the deceived event handler considers `remove 1 2 'X'` as a violation to the read-only range and uses `insert 1 2 'X'` to recover the change, `X` successfully replaces `B` and shows up in the read-only range.

The most straightforward method to avoid this issue is to disable the undo/redo operations. Recovering with a preserved document can solve this issue without disabling undo/redo, but as mentioned just now, this introduces some new problems and was not eventually applied.

Just like the game progress, the player would not want to lose the editing progress

of the answer when he/she quits from a puzzle-solving interface, so the editing progress in the built-in editor should be frequently saved. Using a function taking current text and anchors in the editor and convert them back to a template, this is, the inverse function of the template parsing function, the editing progress is saved as a template file and will be loaded instead of the initial template next time. For example, the text and anchors `AB|C123D|EF` will be converted to `ABC[-123-]DEF` when saved.

Although there is no official implementation of read-only ranges in Ace Editor, a non-official one (denoted by `Impl0`) written by [45] can be found on StackOverflow [22]. Although `Impl0` is originally designed for setting multiples lines to read-only, it can be modified to support arbitrary ranges. Compared with the implementation in HaskellQuest (denoted by `Impl1`), `Impl0` supports using `<editable>` labels to indicate editable lines while `Impl1` uses `[- -]` quotations (see Section 3.4.2) to denote arbitrary editable ranges; both `Impl0` and `Impl1` successfully prevent most changes to the read-only ranges, but Chinese input methods can modify the read-only ranges of both implementations as well; `Impl0` allows the usage of undo/redo while `Impl1` does not, because `Impl0` prohibits changes which violate read-only ranges from happening instead of using change-reverting. For `Impl0`, drag-and-drops can only happen inside an editable range. On the other hand, for `Impl1`, a chunk of text can be dragged from a read-only range and dropped inside an editable range. This action is equivalent with copying the selected text and pasting it inside the editable range, and it is allowed because all the changes to the read-only range caused by drag-and-drop can be completely reverted.

3.4.3 Evaluation environment

To check the correctness of player's code while allowing the existence of multiple answers, HaskellQuest needs to call GHC and evaluate the code submitted by the player. As mentioned in Section 3.2, Electron provides developers with the functionality (a Node.js module [8], actually) of calling other applications and receiving the output from stdout. Thus, as long as GHC is available and can be called by HaskellQuest, all the evaluation and checking work can be handled by GHC and HaskellQuest only needs to collect and analyse the outputs.

The normal process of running a Haskell program is compiling the source code with GHC and run the generated executable files. But in this project, to reduce the steps required to check the correctness of a submitted answer, GHC is called in expression-

evaluation mode [2], which is similar to GHCi.

Sometimes, the code edited by the player is just an expression, such as the example used in Section 3.4.2. Thus, a prefix and a postfix are also included in the code template file to compose a complete variable or function definition. For example, if the code submitted by the player is `not (False && (True || False))`, with a prefix `expr :: Bool\nexpr =` and no postfix, the composed code would be as follows.

```
expr :: Bool
expr = not (False && (True || False))
```

Subsequently, the composed code will be appended to `Main.hs` in the corresponding question folder (see Section 3.4.1) and evaluated by GHC. An expected but desirable feature achieved unintentionally by appending the composed code to the source file is that the player cannot `import` other modules by him/herself, because the `import` declarations must appear at the beginning of a module [58]. The content of a typical `Main.hs` looks like the following:

```
module Main where

import qualified Test.QuickCheck as QC
import qualified GenJSONInfo as J

prop_q :: Bool
prop_q = {- property -}

main :: IO ()
main = do
  r <- QC.quickCheckWithResult
       QC.stdArgs{QC.maxSuccess = 1000, QC.chatty = False}
       (QC.within 2000 prop_q)
  putStrLn (J.genJSONInfo r)
```

As a commonly used property-based testing library, QuickCheck [52] [30] is chosen to verify the correctness of the player's code. It will also monitor the time consumed in case there is an infinite loop or extremely inefficient answer.

The `GenJSONInfo` module is in charge of converting the result from QuickCheck to a JSON object (in the form of a string), so HaskellQuest can parse it without an extra parser. The most important field in the generated JSON object is `stat`, which is a text tag standing for the status of the result of execution. The possible values of `stat` in-

clude "AC" (Accepted), "WA" (Wrong Answer), "TLE" (Time Limit Exceeded), "RE" (Runtime Error), "CE" (Compiler Error) and "INV" (Invalid Status). Some of these status tags are taken from commonly used abbreviations in online judge systems [84]. The first four tags can be generated by analysing the result from QuickCheck, but for "CE", if the player's code does not compile and causes a compiler error, GenJSONInfo has no chance to get the error information and this is handled by the JavaScript side – the compiler error will be captured as an error during the execution of GHC and a fresh JSON object with "CE" tag will be generated. The "INV" tag is only used for an incomplete evaluation, for example, the player quits the puzzle-solving interface before the evaluation completes.

If there is a correct solution named answer to be compared with the player's solution, let's call it result, answer will not be a top-level definition in most cases to avoid that the player defines result using answer. In the code below, with answer defined in the scope of prop_q, the player will not have access to the name answer when defining result.

```
prop_q :: Int -> Bool
prop_q i = result i == answer i
  where answer n = {- definition -}
```

Sometimes, the most suitable question does not involve asking the player to implement something or perform a similar task where multiple solutions exist. Instead, it would be a case where more strict criteria will be taken. The following pseudo-code is an example.

```
   (\x y -> x * 5 - 8) (13 - 3) 21
~> ??
~> 10 * 5 - 8
~> 50 - 8
#> 42
```

The ~> symbol here is inspired by the $\rightsquigarrow$ symbol used in "Haskell: the Craft of Functional Programming" [82]. It can be considered as a reduction operator in small-step operational semantics. And the #> symbol here is a variant of ~> used to denote the final step, which means the expression is reduced to a value. The player will be asked to complete the reduction by replace the ?? symbol with a proper expression, and the rest of the pseudo-code is, of course, read-only.

Under the call-by-need strategy which is the one applied by GHC, the evaluation

process would be as follows.

```
   (\x y -> x * 5 - 8) (13 - 3) 21
~> (13 - 3) * 5 - 8
~> 10 * 5 - 8
~> 50 - 8
#> 42
```

On the other hand, if the player applies the call-by-value strategy, his/her answer would probably be the following one.

```
   (\x y -> x * 5 - 8) (13 - 3) 21
~> (\x y -> x * 5 - 8) 10 21
~> 10 * 5 - 8
~> 50 - 8
#> 42
```

If the evaluation process under the call-by-need strategy is the expected answer, it should be distinguished from the call-by-value one. Plain text comparison will not work because there might be extra spaces or other ignorable characters, and the comparison should happen on the token or AST (Abstract Syntax Tree) level. But implementing a Haskell parser would add unwanted complexity to this project. Thus, Template Haskell [76] is used to accomplish this goal.

As its name suggests, Template Haskell is a template system for Haskell. As a GHC extension [1], Template Haskell allows compile-time meta-programming [76], or the generation and manipulation of Haskell code at compile-time. A typical application of Template Haskell can be found in the famous lens package [23], where Template Haskell is used to construct lenses for user-defined datatypes automatically [12].

Shown below is an exemplar `Main.hs` using Template Haskell.

```haskell
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}

module Main where

import qualified Test.QuickCheck as QC
import qualified Test.QuickCheck.Monadic as QCM
import qualified GenJSONInfo as J
import qualified Language.Haskell.TH as TH
import qualified Language.Haskell.TH.Alpha as THA

```

```
prop_q :: QC.Property
prop_q = QCM.monadicIO $ do
  a <- QCM.run (TH.runQ answer)
  r <- QCM.run (TH.runQ result)
  c <- QCM.run (THA.expEqual a r)
  QCM.assert c

main :: IO ()
main = do
  r <- QC.quickCheckWithResult
       QC.stdArgs{QC.maxSuccess = 5, QC.chatty = False}
       (QC.within 2000 prop_q)
  putStrLn (J.genJSONInfo r)

infixr 0 ~>
(~>) = (:)

infixr 0 ~~~>
(~~~>) = (:)

infixr 0 #>
(#>) l x = l : [x]


answer = [|
     (\x y -> x * 5 - 8) (13 - 3) 21
  ~> (13 - 3) * 5 - 8
  ~> 10 * 5 - 8
  ~> 50 - 8
  #> 42
|]
```

A new symbol `~~~>` is introduced to represent multi-step reduction. All three symbols are defined as right-associative list construction operators with the lowest precedence. Thus, the evaluation process is now a list of expressions. The variable `answer` is the expected answer and is quoted with expression quasi-quote brackets `[|  |]` [76], which will convert the quoted Haskell code into an value of type `Exp` (a data-type representing Haskell expressions) inside the `Q` monad (`Q` for quotation), and the type `Q Exp` is also referred as `Expr` in [76]. For example, putting the `Q` monad aside, a quoted expression `[| (\x y -> 3 * (x + y)) |]` is converted to the following `Exp` value.

```
LamE [VarP x_0, VarP y_1]
  (InfixE
    (Just (LitE (IntegerL 3)))
    (VarE GHC.Num.*)
    (Just (InfixE
      (Just (VarE x_0))
      (VarE GHC.Num.+)
      (Just (VarE y_1)))))
```

A value of type `Exp` can be treated as the AST of an expression, and we can easily quote the player's code and bind it to a variable (`result` here) with pre- and post-fix in code template, so it seems that the only thing left to do is extract the `Exp` values from `answer` and `result` and check their equivalence. However, a naive comparison with `(==)` will not work because Template Haskell is lexically scoped and it may rename some variables in a quoted expression. Thus, what actually needs to be checked is the α-equivalence of the expressions inside `answer` and `result`. A library [39] for alpha-equivalence in Template Haskell is used here.

3.5 Background story for HaskellQuest

The story is one of the important elements that make HaskellQuest a text adventure game. Beside making the use of Haskell and GHC look reasonable and guiding the player to explore different aspects of Haskell, the story is also expected to be attractive and logically consistent.

As mentioned in Section 3.3, the story is divided into chapters. The first chapter is named `Prelude`. This name is taken from the Haskell standard module which is imported by default. It also indicates that this chapter introduces some background settings. The main story is developed around a project named CUBIC (Crystal Utilisation Based on Interdimensional Communication), which is a cooperation between two similar worlds in different dimensions – Earth and Thyroc. The following italicised text is a summary of the background.

Six years ago, mysterious crystals began to appear on Thyroc. These crystals were collected and analysed by the scientists. After countless complex experiments, some crystals broke down via a process called sublimation and turned into luminous particles named ponectors, which were found to be safe and high-density power sources.

However, the positions of ponectors were not stable. They showed the ability of

jumping randomly between two worlds and also attracted the attention of Earth scientists. Researchers from both worlds developed their own technologies of stabilising the ponectors, and in this process, both worlds realised the existence of each other. After that, researchers from both worlds worked together and constructed an interdimensional power and communication network of ponectors. In the meantime, researchers also found that the generation of crystals on Thyroc was correlated with the amount and activity of Haskell projects on Earth. Subsequently, thanks to the co-operation of scientists from both worlds, devices called sublimators were invented to accelerate the sublimation of crystals. A sublimator can analyse a crystal and present it as a Haskell puzzle, and the solution to the puzzle will be converted to an operation sequence on the crystal. If the solution is correct, the crystal will sublimate and the generated ponectors will be collected. The crystal virtualisation technique was also developed at this moment. A virtual crystal can simulate the behaviour of a real crystal, but it will not produce real ponectors.

The CUBIC project started two years ago. The members of CUBIC are from both Earth and Thyroc, they will work in pairs to collect more ponectors. For each pair, they share a common crystal storage, but only the member from Earth can authorise the usage of a sublimator. Virtual crystals are also used in CUBIC to let the members practise before sublimating real crystals.

Some analogies in the background, such as the sublimator standing for the complier, and the ponectors standing for the reward, are relatively obvious. Besides, the properties of a puzzle (see Section 3.4.1) are also explained in the story. For example, what corresponds to the reward reduction system in the story is that wrong operation sequences from the sublimator will distort the internal structure of a crystal and reduce the production of ponectors.

Two NPCs from Thyroc, Dr Wilson and Alvin, together with the avatar of the player in this story, Harley, from Earth, are involved in the CUBIC project. Dr Wilson is in charge of introducing the concepts in Haskell and guiding the player. Alvin works together with the player and will also do part of the tasks. This design can balance the workload of the player and the NPC, and was mentioned in my IPP [54]. According to the background story, only the member from Earth can authorise the usage of a sublimator, so Alvin will ask the player to sublimate his crystals, and the player can get some extra ponectors without solving puzzles. In HaskellQuest, this feature is implemented by storing answers as editing progress files in the question folders belonging to the NPC.

As an easily ignored third-party character, the creators of these crystals will also be involved in the story. And their true purpose of sending crystals to Thyroc may lead the currently peaceful story to another direction.

At the end of `Prelude`, Alvin gets a real crystal from his friend and abruptly leaves without explanation. This is the turning point of story, and before Alvin comes back, the player can only collect ponectors by him/herself in the next chapter named `Chapter 1`.

The are several possible directions of future development. A preferred one is the fight against the ponector creator. But due to time limitation, the rest of the story is yet to be written.

The writing process of this story is very different from writing a normal article, because the player's choices can decide the development of the story. Moreover, if each conditional branch creates two divergent storylines, the number of storylines will grow exponentially and the story files will become extremely hard to maintain very soon. Thus, to simplify the story and save time, all the branches in currently available chapters are merged into no more than five sections.

3.6 HaskellQuest as a playable Haskell book

The design goal of HaskellQuest is a playable Haskell book, so the way in which the structure of HaskellQuest is organised is also similar to that of a book. According to the minimal implementation (see Section 1.1), the concepts covered in HaskellQuest should at least include expressions, function declarations, types, lists, list comprehensions and recursive definitions. The major books being referenced are "Learn You a Haskell for Great Good!" [56], "Haskell: the Craft of Functional Programming" [82] and "Real World Haskell" [63]. To avoid one of the problems of ScalaQuest, that is, the lack of explanations, every puzzle in HaskellQuest is presented with a detailed description and examples. The notation `[- -]` (see Section 3.4.2) will be used in this section to denote editable ranges.

Depending on the choice of the player, several puzzles demonstrating the usage of the puzzle-solving interface, or sublimator according to the background story, may be presented after introducing the background. The player can get familiar with the available interactions and the interface logic of HaskellQuest with these puzzles, which are only for demonstration and not related with any Haskell concepts.

After that, the player will meet the first Haskell puzzle in this game. Instead of

starting with arithmetic operations as in [56] and [63], HaskellQuest starts with the introduction to Boolean values (`True` and `False`) and logic operators (`not`, `&&` and `||`), because Boolean values can probably be regarded as the most simple form of values in Haskell besides `() :: ()`. Nevertheless, numbers and basic arithmetic operators (`+`, `-`, `*` and `/`) are introduced right after Boolean values. A typical puzzle is replacing the `??` in

```
[-??-] 2 [-??-] 3 [-??-] 1 [-??-] 7 [-??-]
```

so the expression evaluates to `42`.

Since the player has already seen and done computation with values of different types, the concept of type is introduced afterwards. The definition of type is kind of tricky because the meaning of the word "type" varies in different contexts [65]. HaskellQuest adopts a simplified version of the definition in "Types and Programming Languages" [67], and the type of an expression is defined as the collection of possible results it can evaluate to. Compared with other definitions [65], this definition is more suitable for Haskell. Puzzles related with types make use of the fact that Haskell is a statically typed language [55]. The player will be asked to add type annotations for expressions such as `(7 + (19 :: Float)) :: [-??-]`, and there is no need to run actual random tests because the type checking runs at compile-time and any type error will lead to a `"CE"` status tag.

The comparison operators (`==`, `/=`, `<`, `>`, etc.) are introduced subsequently. In this part, the players will see how Boolean values and numbers are mixed in an expression by solving a relatively easy puzzle. Meanwhile, a special task is given to the NPC Alvin, that is, computing the mean value and the variance of a set of sample values and checking if they are in the expected ranges. This task seems to be easy, and the mean value part is indeed straightforward to write, but for the variance part where the mean value is subtracted from each sample, the expression of mean value need to be repeated many times because no way has been introduced to keep an intermediate result. After showing this puzzle to the player, Alvin will complain about this puzzle and ask if there exists some sort of "container" to hold a value. This is the right moment to introduce variables in Haskell. Although the common container analogy for variable is used here, variables in Haskell are actually identifiers or names rather than containers. The player will also be notified about this, and a puzzle affected by the example code in [64] will be used to demonstrate that multiple declarations of the same variable are not allowed.

The next part of HaskellQuest is the introduction of functions, and this is a special — probably the most special — part in HaskellQuest. Instead of using the black-box

analogy [83], or starting with simple function definitions [57], lambda abstraction is introduced first in HaskellQuest.

By giving the syntax of defining lambda abstractions in Haskell [58] and comparing them with mathematical functions, lambda abstractions are introduced to the player as a special kind of values which can process values. This definition eliminates the boundary between functions and other values, and there is no need to explicitly introduce the concept of higher-order functions in this case. The basic idea of higher-order functions is indicated by the definition: functions are values that process values, functions are values, so functions can process functions.

This arrangement is also affected by the GHC `Core` language which is used as the intermediate representation of programs in GHC [59]. It is an implementation of System FC [80], and functions in Haskell source code will be converted to explicitly typed lambda abstractions in `Core`. For example, the following Haskell code defines an `add` function adding two integers.

```
add :: Int -> Int -> Int
add x y =  x + y
```

And this is the `Core` code generated by GHC (`-O0` option is used to turn off optimisation, otherwise the function is optimised to `(+)`).

```
add :: Int -> Int -> Int
add = \ (x :: Int) (y :: Int) -> + $fNumInt x y
```

The type signature of `add` does not change, but its parameters are moved into a lambda abstraction.

A typical puzzle at this stage has already been used as an example in Section 3.4.3, that is, producing the evaluation process with the call-by-need strategy for a function application. The player will also be given an example showing a relatively complex evaluation process like the following one to reinforce his/her memory about functions as values.

```
   (\f a b -> f (a * a) b) (\x y -> x - y) 5 ((\x -> x + 1) 7)
~> (\x y -> x - y) (5 * 5) ((\x -> x + 1) 7)
~> (5 * 5) - ((\x -> x + 1) 7)
~> 25 - ((\x -> x + 1) 7)
~> 25 - (7 + 1)
~> 25 - 8
#> 17
```

The idiomatic and usual way of defining a function in Haskell is, of course, introduced later, together with the syntax of function types. But for a smooth transition from lambda abstractions, the function binding syntax [58] is introduced as another way to write a variable bound to a lambda abstraction. For example,

```
square :: Int -> Int
square x = x * x
```

is just an easier and clearer way to write

```
square = (\x -> x * x) :: Int -> Int
```

After the function definition part, the player will be asked about the number of different type annotations that can be assigned to the lambda abstraction `(\x -> x)` assuming that besides lambda abstractions, there are only values of type `Int` to compute with (related arithmetic operators are also available as constants). There are two options for the player to choose, "only one" and "infinitely many". No matter which one the player chooses, examples of function types such as

```
((\x -> x) :: (Int -> Int) -> (Int -> Int)) ((\x -> x) :: Int -> Int)
```

or

```
((\x -> x) :: (Int -> Int -> Int) -> (Int -> Int -> Int))
  ((\x y -> x + y) :: Int -> Int -> Int)
```

will be given and by recursively constructing similar function applications, the lambda abstraction `(\x -> x)` can have infinitely many possible type annotations. Then the player will be notified about the uniform behaviour of `(\x -> x)` for input of any type [66]: it does not care about the exact argument type, and it always simply returns its argument as it is. Thus, by introducing a type variable `t0`, we have a general identity function `(\x -> x) :: t0 -> t0` whose type annotation means this function accepts an argument of type `t0`, which could be any type, and its return value has type `t0` as well.

The purpose of this part is introducing parametric polymorphism in Haskell and now the learning path is actually clear. In HaskellQuest, the player will be introduced to untyped lambda calculus [68], simply typed lambda calculus [69] and System F [70] sequentially. These terms are not going be mentioned in the game to avoid obscurity. The player only needs to have a basic understanding of untyped lambda calculus, and the other two are just introduced conceptually.

At the end of `Prelude`, a timed puzzle about the Church Numerals [71] will be

presented, which is a relatively hard one and its title in the game is `Church Numerals`. The player will be asked to complete the evaluation process of `mult two three (\n -> n + 1) 0` where `mult`, `three` and `two` are defined as follows.

```
two :: (a -> a) -> a -> a
two = (\f x -> f (f x))

three :: (a -> a) -> a -> a
three = (\f x -> f (f (f x)))

mult :: ((a -> a) -> a -> a) ->
        ((a -> a) -> a -> a) ->
        ((a -> a) -> a -> a)
mult a b = (\f x -> a (\z -> b f z) x)
```

By solving this puzzle, the player can practise β-reduction [72] in lambda calculus and learn about another representation of natural numbers.

The next chapter named `Chapter 1` is mainly about lists and recursion. The list is brought to the player in a traditional way as a collection of values of the same type. And the list comprehension is introduced by comparing it with the set-builder notation in mathematics. A similar approach is also used in [57].

The puzzle designed for list comprehension (puzzle title: `Virtual Crystal (List - 10)`) is a relatively tricky one. The player will be asked to generate Pythagorean triples using Dickson's method [75]. The corresponding algorithm and a code template using list comprehensions will be given, and the player's task can be simplified to finding all the non-repetitive factor pairs of an integer. For example, a list of the factor pairs of `24` is `[[1, 24], [2, 12], [3, 8], [4, 6]]` (the order does not matter). The factor pairs are also represented with lists because tuples are not introduced yet. The tricky part is that, with respect to the generation of factor pairs using list comprehension, there is a straightforward but slow method as follows.

```
pairs n = [[a, b] |
  a <- [1..n],
  b <- [1..a],
  a * b == n]
```

And a more efficient one with the same code structure.

```
pairs n = [[a, b] |
  a <- [1..floor (sqrt (fromIntegral n))],
  b <- [quot n a],
  mod n a == 0]
```

Since there is a execution time limitation in the evaluation environment (see Section 3.4.3), if the player's code is not efficient enough, it will not be accepted even if it is theoretically correct.

Subsequently, after learning about some condition-related syntax such as guards and pattern matching [58], the player will be introduced to the concept of recursion. Thanks to the notations defined in Section 3.4.3, they can still be used to demonstrate the working principle of recursion. As long as the player feels comfortable about showing the step-by-step evaluation process, understanding the basic idea of recursion should not be too hard. For example, the following code shows the definition of an `append` function working in the same way as `(++)` and the evaluation process of `append [1, 2, 3] [4, 5]`.

```
append [] x = x
append (x : xs) ys = x : append xs ys

   append (1 : 2 : 3 : []) (4 : 5 : [])
~> 1 : append (2 : 3 : []) (4 : 5 : [])
~> 1 : 2 : append (3 : []) (4 : 5 : [])
~> 1 : 2 : 3 : append [] (4 : 5 : [])
#> 1 : 2 : 3 : 4 : 5 : []
```

The player will practise by showing the computation process of the fifth number in the Fibonacci sequence [50] and defining his/her own version of the `length` function [58] (puzzle title: `Virtual Crystal (Recursion - 2)`).

Due to time limitations, the story in `Chapter 1` is incomplete and more puzzles are to be added.

Chapter 4

Result and evaluation

This chapter analyses the results and user evaluation of this project. A presentation of the look and performance of the game can be found in Section 4.1. The user evaluation of HaskellQuest and related discussion can be found in Section 4.2.

4.1 Self-evaluation

The major development machine of HaskellQuest was a Lenovo laptop running Windows 10. The Windows version of HaskellQuest was developed and (mainly) tested on this laptop. Besides, a MacBook Pro running macOS High Sierra was used to develop and test the platform-specific part of the macOS version of HaskellQuest. Due to time limitations, no Linux version was released. In principle, there should be no problem in producing a Linux version because Electron supports Linux packaging.

Figure 4.1 shows the messaging and puzzle-solving interface of HaskellQuest. Figure 4.2 shows the main screen (including title text, chapter selector and question list) and the option buttons. The prototype design shown in Section 3.1 is also included as a comparison in Figure 4.3. All these screenshots were taken from the development machine.

Figure 4.1: A screenshot of HaskellQuest showing the puzzle-solving interface and messaging interface

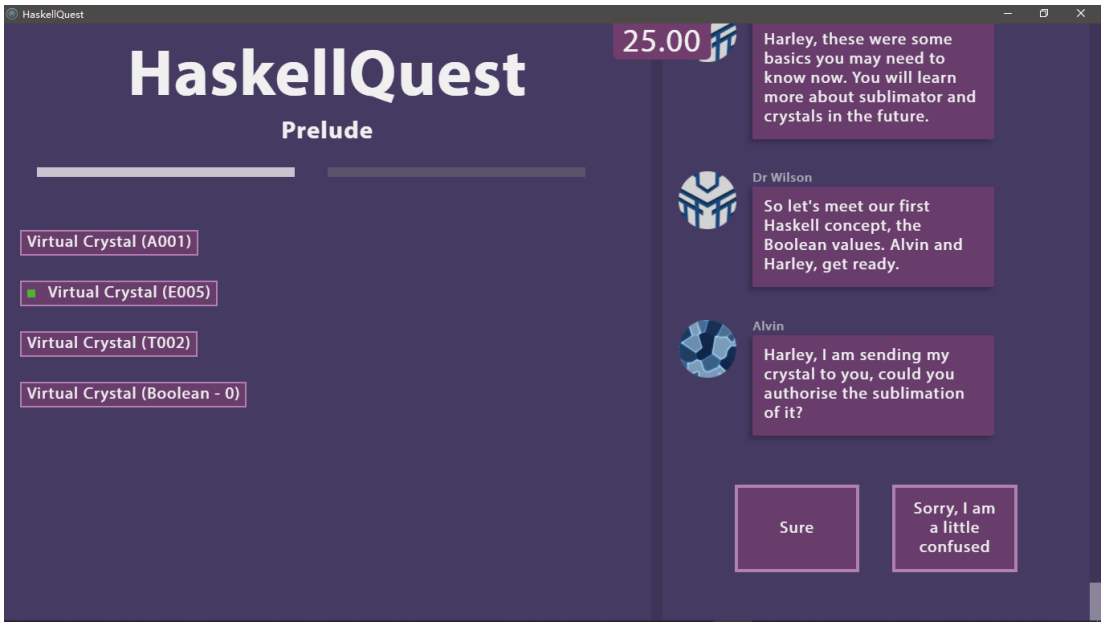

Figure 4.2: A screenshot of HaskellQuest showing the question list and option buttons

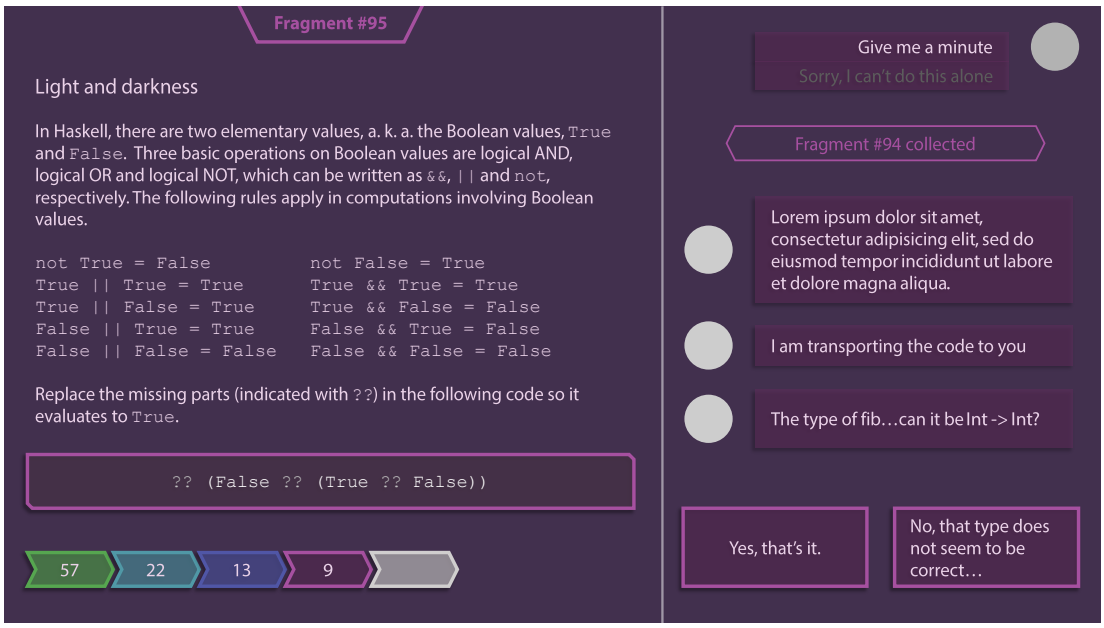

Figure 4.3: The prototype UI design of HaskellQuest

As expected (see Section 3.3), the messaging interface can sequentially display messages extracted from story files with smooth animation, and jump to different sections according to the player's choices. Besides, the game progress can be saved and recovered correctly.

The puzzle-solving interface also works well. Descriptive text and code are shown in different fonts and styles, and the built-in editor can display code with syntax highlighting and handle read-only ranges properly. A known problem mentioned in Section 3.4.2 is that Chinese input methods can break the read-only ranges. Another problem found later during the test is that copy-and-paste with `Command-C` and `Command-V` in the built-in editor does not work on macOS. Due to time limitations, the reason and potential solution for this problem is still unknown.

Some of the problems in ScalaQuest [33] (see Section 2.4) are also avoided in Haskell-Quest. Each puzzle is displayed together with a detailed explanation; the evaluation environment naturally allows the existence of multiple answers; and an execution time limitation is set for each puzzle to prevent infinite loops.

During the test, a typo in a code template was noticed: the **#>** symbol indicating the final step of an evaluation process was mistakenly written as **~>**. The following pseudo-code is an example.

```
   (\x -> x + 1) 3
~> 3 + 1
#> 4
```

is written as

```
   (\x -> x + 1) 3
~> 3 + 1
~> 4
```

In Haskell source code, this typo should lead to a type error, but actually, it will not cause any problem, because the whole evaluation process is wrapped by an expression quasi-quote (see Section 3.4.3) which converts the quoted expression to an AST without type-checking, and this typo does not significantly affect the structure of the generated AST (~> and #> have the same fixity and precedence).

This typo seems to be just a minor mistake, but it does reveal the possibility that there could be hidden mistakes in code templates and correct answers. The solution is straightforward: typed expression quasi-quote `[|| ||]` in Template Haskell [1] will type-check the quoted expression and the whole quotation will have type `Q (TExp t)` where `t` is the type of the quoted expression. Besides avoiding similar typos, typed expression quasi-quote can help check if the quoted evaluation process is correct, because evaluation should preserve type [73]. Actually, typed expression quasi-quote was used in an earlier implementation of evaluation environment, but it was replaced by ordinary expression quasi-quote to simplify the code, because type information will be removed while checking the α-equivalence of two expressions.

As for the puzzles and Haskell concepts involved in this game, all the required concepts (expressions, function declarations, types, lists, list comprehensions and recursive definitions) in the minimal implementation (see Section 1.1) are covered. There are two chapters, `Prelude` and `Chapter 1`. Due to time limitation, `Chapter 1` is incomplete. There are 3 demonstration puzzles, 10 puzzles for the player and 6 puzzles for the NPC in `Prelude`, and 5 puzzles for the player in `Chapter 1`. There are two timed puzzles, `Church Numerals` at the end of `Prelude` and `Virtual Crystal (Recursion - 2)` at the end of `Chapter 1`.

4.2 User evaluation

As a serious game, HaskellQuest is expected to be both educational and entertaining, but it is hard to find an objective standard to evaluate the effects of teaching or entertainment. Thus, collecting and analysing feedback from real players is necessary.

Due to the limitation of time and available players, only 10 prospective first-year undergraduate students were involved in the testing and until the deadline on 12th August, 5 of these students submitted their feedback via an online survey form. The whole process of user evaluation was anonymous and GDPR-compliant, and was approved by the Informatics ethics process.

The questions on the online survey form and related discussions are listed below.

1. **Please input the link of HaskellQuest_Report.json here.**

   Each player was asked to upload a report file generated by HaskellQuest to an anonymous file sharing service [16], and then submit the generated download link in this survey question.

   A report file is the combination of all the saved progress files. In other words, it provides the following information.

   - The game progress (chapter index, message index, total reward, etc.).
   - The player's choice at each conditional branch.
   - All of the player's failed attempts and the finally successful one in each puzzle.
   - The valid time the player spent on each timed puzzle. The time consumed to complete a timed puzzle without quitting the puzzle-solving interface is considered valid.

   Some noteworthy or interesting results are listed below. These results will be analysed in combination with other survey questions later in this section.

   - None of the players completed all the available chapters. Three players completed `Prelude`, and stopped at `Virtual Crystal (List - 10)` in `Chapter 1`, which is the Pythagorean triple puzzle mentioned in Section 3.6. The other two players stopped at `Church Numerals` at the end of `Prelude`.
   - No valid time information for timed puzzles was collected. This means all these players quit the puzzle-solving interface at least once when they tried to solve `Church Numerals`.

- Most players tried many times when they got stuck on a puzzle. For example, the average attempts count is 46 for `Virtual Crystal (List - 10)`.
- Most puzzles were not hard for these players. The average attempts count is 1.81 for other puzzles except `Church Numerals` and `Virtual Crystal (List - 10)`.
- Most puzzles solved by the NPC were submitted without any modification.
- Four players chose "No" and one player chose "Yes" when a NPC asked if a number and a Boolean value can be added together. This question was asked before the concept of type was introduced to the players.
- Two players chose "Only one" and three player chose "Infinite many" in the question about polymorphism mentioned in Section 3.6.

2. **Please select the operating system on which you installed HaskellQuest.**

   This question is about the operating systems used by the players. Three players used Windows and two players used macOS to play HaskellQuest. This question was based on the concern that different platforms may lead to different experiences. But according to the final result and other survey questions, the platform is not a key factor because there was no significant difference between the results of users using the two platforms.

3. **How will you rate HaskellQuest as a game prototype?**

   This is the overall rating of HaskellQuest. The following table shows the result.

Rating	Very Poor	Poor	Fair	Good	Very Good
Quantity	0	0	2	2	1

   Generally, HaskellQuest seems to be rated as a good game prototype, but many factors contribute to this rating, and the two "Fair" ratings indicate that there are some unsatisfactory parts. The remainder of this section will discuss the good and unsatisfactory aspects of HaskellQuest.

4. **How often did the user interface of HaskellQuest get stuck/become unresponsive/crash?**

   This question is about the stability and smoothness of the interface of HaskellQuest. The following table shows the result.

Frequency	Never	Rarely	Sometimes	Usually	Always
Quantity	3	2	0	0	0

Thanks to the React framework [31] and asynchronous processing (see Section 3.2), the survey result strongly supports that the interface of HaskellQuest is smooth and stable. Having a responsive UI is very important for HaskellQuest, because the player will not expect a text-based game like HaskellQuest to have complex and realistic visual effects, and HaskellQuest indeed has a relatively simple UI, lowering the player's tolerance to choppy animations or transitions.

5. **"The background story of HaskellQuest is interesting."**

   This question is about the players' interest on the background story of HaskellQuest. The following table shows the result.

Attitude	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Quantity	0	1	1	2	1

This divergent result is as expected because different people have different thoughts and preferences, but generally it shows that the background story of HaskellQuest is not too bad. Beside the story setting and the writing skill, the way in which the story is presented is also a factor affecting the experience of readers: if a player is not satisfied with the messaging interface, he/she is not likely to enjoy the story.

The problem mentioned in Section 2.4 still exists in HaskellQuest, that is, Haskell is not an essential or irreplaceable element in the story. Although according to the story in Section 3.5, Haskell is used as the language of sublimators, and the activity of Haskell projects on Earth can affect the generation of crystals on Thyroc, Haskell can still be replaced by another programming language and the story would be the same. On the whole, the development of the story in HaskellQuest is not tightly bound with the characteristics of Haskell, and this should be improved in the future.

6. **"The puzzles in HaskellQuest are well-organised."**

   This question is about the design and organisation of puzzles in HaskellQuest. The following table shows the result.

Attitude	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Quantity	0	0	3	2	0

An important fact not shown in the table is that, the three players choosing "Neither agree nor disagree" are also the three player who completed `Prelude` and stopped at `Virtual Crystal (List - 10)` in `Chapter 1`, and the two players choosing "Agree" are also the two player stopped at `Church Numerals` in `Prelude`.

This makes the result easier to explain, because the organisation of puzzles in `Prelude` is indeed better than `Chapter 1`.

In `Prelude`, Haskell concepts are introduced in the sequence of Boolean values, numbers, types, comparison operators, variables, lambda abstractions, function definitions and parametric polymorphism. More details about their inner connections can be found in Section 3.6. Each concept is related with one or two simple puzzles, and the whole chapter ends with a relatively hard puzzle `Church Numerals`.

On the other hand, at the beginning of `Chpater 1`, lists in Haskell are introduced, but the need of lists is not well-motivated. Besides, list syntax, arithmetic sequences [58] and list-related functions are mixed in the description of a single puzzle for lists. The introduction to list comprehensions is not mixed with other concepts, but the corresponding puzzle might be too hard for Haskell beginners.

A possible improvement would be to decompose the current list puzzle into three or more puzzles, only introducing one concept in each of them. For list comprehension, the complexity of the corresponding puzzle should be reduced; maybe a simple filtering with list comprehension would be enough. Given that list comprehension is just another way of writing programs using list monads, the exploration of its working principles can be put in a later stage.

7. **"I have learned something new about Haskell by playing HaskellQuest."**

This question is about the subjective feelings of the players whether they have learned new stuff by playing this game. The following table shows the result.

Attitude	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Quantity	0	0	1	1	3

The result is encouraging because three players selected "Strongly Agree", indicating that the educational purpose of HaskellQuest was achieved.

Actually, the process of learning can also be observed in the message list. According to the summary of collected report files (see the first survey question), not all the players chose the right answers to NPC's questions, which means that the right answers are not obvious. And the players, especially those choosing the wrong answers, can learn by reading successive messages or solving related puzzles. For example, a player chose "Yes" when a NPC asked if a number and a Boolean value can be added together. Subsequently, this player was asked to correct the code `42 + [-False-]`, and after several attempts, he/she eventually understood and solved this puzzle.

8. **"I want to try more chapters of HaskellQuest."**

   This question is about the subjective feelings of the players that whether they want to try more chapters of HaskellQuest. In other words, the result of this question will reveal the attractiveness of HaskellQuest to these players. The following table shows the result.

Attitude	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
Quantity	1	0	1	1	2

The result is divergent, but two "Strongly agree" still indicate that HaskellQuest attracted some players. A noteworthy point is that one player selected "Strongly disagree", and this player also expressed in the feedback that he/she was not satisfied with the code editor and messaging interface of HaskellQuest, which could be a possible reason for this negative attitude.

9. **Please input here the title of the hardest puzzle in HaskellQuest in your opinion.**

For this question, three players selected `Virtual Crystal (List - 10)`, the other two selected `Church Numerals`, which are exactly the puzzles where they got stuck.

`Church Numerals` is a relatively hard puzzle compared with other puzzles in `Prelude`, and the player needs to be careful when writing the answer because there are lots of parentheses, but most players should be able to solve it if given more hints.

As for `Virtual Crystal (List - 10)`, requiring correctness and efficiency at the same time, this puzzle could be too hard for Haskell beginners. One of the players provided a naive solution for this puzzle (see Section 3.6), but still failed to write a more efficient one.

10. **Do you have any suggestions or other feedback?**

Two players mentioned in their feedback the need for hints. The hint system is part of the initial plan [54], but it is not implemented due to time limitations. There are many things to consider when designing a hint system, such as when a hint should be given, what its content should be, and how much it should cost if the player needs to use collected reward to "buy" hints.

Another players mentioned that the timings for the messages to appear were strange. This could be a problem, but from another perspective, it could also be a design choice (that requires further improvements). To create the experience of a real-time text adventure in HaskellQuest, the sending delay of a message is tuned to be the approximate time that a real person needs to compose that message using a keyboard. Actually, simulating the delay caused by keyboard typing is not the only option; for example, Lifeline: Whiteout [25] simulates the delay of a speech-to-text device. In Lifeline: Whiteout, a message is normally a short sentence, which means the messages will be sent with relatively uniform delays and the player will have enough time to finish reading the current message before the next one comes in. Besides, there are lots of branches in Lifeline, providing the player with extra time to stop and read all previous messages carefully. But things are different in HaskellQuest. Firstly, many messages in HaskellQuest are fairly long. If a long message is followed by a very short one, the message list will be updated before the player can finish reading the long message. And things can get more complex when there are multiple NPCs sending messages at the same time. However, if we want to leave enough time for the player to finish reading each message, we will need to

intentionally add the corresponding delay at the end of each message, and this is not the simulation of real life. If we insist on simulating message sending and receiving in real life, breaking down long messages and finer adjustments to the delays might improve the experience.

A player complained that the visible area of the built-in code editor is too small and the message list always scrolled to bottom automatically when a new message appeared regardless of whether he/she was reading previous messages, making the game experience unnecessarily frustrating and annoying. The visible area of the built-in editor is currently constrained by the area of the puzzle-solving interface, so a floating editor might improve the code editing experience. For the message list, scrolling to bottom is indeed a convenient but inelegant solution to notify the player that a new message has arrived, and showing a notifier could be a better choice.

Chapter 5

Conclusion

In this project, a prototype of a cross-platform serious text adventure game named HaskellQuest teaching functional programming in Haskell is implemented.

With a background story happening in another world and guided by the NPCs, the player explores different Haskell concepts and solves corresponding puzzles in HaskellQuest. During this process, the player can send messages to the NPCs by clicking on different options appearing in the message list and the player's choices will affect the development of the story. Besides, just like in a Haskell book, every concept in HaskellQuest comes with examples and detailed explanation, and during the game, the player can freely review all previous messages and puzzles he/she has done.

According to the self-evaluation and user evaluation, it is practical to teach Haskell with such a game and the players can actually learn things. The whole framework built in HaskellQuest is extensible and more functionalities, story chapters and puzzles can be added in the future. A possible application of this game is to assist the teaching of Haskell in universities. Programmers can also use this game to learn Haskell or to practice their skills.

Some problems are also observed in HaskellQuest, such as the lack of a hint system and the unsatisfactory story structure. Some potential solutions are proposed in Section 5.2.

5.1 Achievements

The following achievements are accomplished in this project.

- A messaging interface displaying text messages and clickable options and a

puzzle-solving interface involving a stylised puzzle description and code editor are implemented. They both work smoothly and stably.

- A story data format encoding the development and branching of a story is designed and used to store the background story of HaskellQuest.
- A code editor where users can only edit the text inside editable ranges is implemented based on Ace Editor. A quotation notation is provided for the developer to define arbitrary editable ranges.
- A Haskell code evaluation environment is implemented. Both multiple answers to the same question and strict code comparison are supported.
- A background story is written to combine text adventure and Haskell education, and it is presented in the messaging interface.
- 24 Haskell puzzles with detailed descriptions are designed.
- The game sequentially covers the concepts of values, expressions, types, variables, lambda abstractions, function definitions, parametric polymorphism, lists, lists comprehensions and recursive definitions.
- A user evaluation was carried out at the final stage of this project. According to the user evaluation, most players gave positive feedback and agreed that they have learned new knowledge about Haskell by playing HaskellQuest.

5.2 Future work

HaskellQuest is just a game prototype, and there is still a long way to go before it can be truly treated as a complete serious game. Lots of work could be done in this process and the remainder of this section introduces some possible directions.

The hint system mentioned in Section 4.2 is an important direction of future work. Technically, loading and displaying hints for a puzzle is not complex. The truly complicated parts are deciding the right moment to display a hint and choosing the most suitable hint for the player. Both parts require HaskellQuest to gather more information about the status of the player, such as the time consumed in the current interface, the areas being most frequently edited and the messages being rechecked. Other components of HaskellQuest might also be modified to support the hint system. For example,

a user-friendly way to display a hint could be that, if the player makes a mistake in his/her code and asks for a hint, the evaluation environment will locate the mistake so the hint system can choose a suitable hint, and the code editor will show the hint as a piece of floating text beside the mistake. Template Haskell would be very useful in this case because it provides tools for analysing the player's code and comparing it with the expected answer.

The UI of HaskellQuest can also be improved. Some potential improvements have already been discussed in Section 4.2. Changes in the UI will not alter the main program and are relatively lightweight in most cases, but the design of the UI (colour schemes, layouts, animations, etc.) is subtle and more user evaluations are required to achieve a satisfactory result.

More chapters should be added to HaskellQuest because for functional programming in Haskell, recursive definitions are just the very beginning. Type classes can be introduced earlier together with parametric polymorphism, so there will be no need to avoid type constraints in code and the player can have better understanding of some concepts such as arithmetic sequences (the type of values in an arithmetic sequence must be an instance of `Enum`). As mentioned in Section 4.2, the list comprehension puzzle might be too hard for Haskell beginners and it should be replaced by an easier one; and the list puzzle will be decomposed to multiple puzzles. After the player gets familiar with lists and recursion, the algebraic data types could be introduced to help the player understand the internal structure of a list, and then more higher-order functions on lists (`foldr`, `foldl`, `zipWith`, etc.) could be introduced. After that, the concept of functor could be revealed to the player as a generalisation of `map`.

As for the story in HaskellQuest, more interactions between the player and NPCs should be added, and to avoid choking the player with numerous Haskell concepts, there should be some breaks allowing the player to experience the entertaining aspects of this game. In the current story, the NPC Alvin has not come back since he left at the end of `Prelude`, and this hanging storyline could be the start of adding more adventure elements to the story. And to strengthen the connections between the story and Haskell, concepts or language features that make Haskell different from other programming languages should be involved in the story. Type classes and monads could be competent candidates for this.

Bibliography

[1] 10.1. Language options — Glasgow Haskell Compiler 8.4.3 User's Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#template-haskell. Accessed: 2018-12-08.

[2] 7.1. Using GHC – Glasgow Haskell Compiler 8.4.3 User's Guide. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/using.html#expression-evaluation-mode. Accessed: 2018-07-08.

[3] About Electron | Electron. https://electronjs.org/docs/tutorial/about. Accessed: 2018-01-08.

[4] Ace - The High Performance Code Editor for the Web. https://ace.c9.io/. Accessed: 2018-05-08.

[5] Android Messages – Apps on Google Play. https://play.google.com/store/apps/details?id=com.google.android.apps.messaging&hl=en_GB. Accessed: 2018-01-08.

[6] Bitbucket | The Git solution for professional teams. https://bitbucket.org/. Accessed: 2018-12-08.

[7] Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org/. Accessed: 2018-01-08.

[8] Child Process | Node.js v10.8.0 Documentation. https://nodejs.org/api/child_process.html#child_process_child_process_execfile_file_args_options_callback. Accessed: 2018-07-08.

[9] Chromium - The Chromium Projects. https://www.chromium.org/Home. Accessed: 2018-01-08.

[10] CodeCombat - Learn how to code by playing a game. https://codecombat.com/home. Accessed: 2018-17-08.

[11] CodeMirror. https://codemirror.net/. Accessed: 2018-05-08.

[12] Control.Lens.TH. http://hackage.haskell.org/package/lens-4.17/docs/Control-Lens-TH.html. Accessed: 2018-15-08.

[13] Daring Fireball: Markdown. https://daringfireball.net/projects/markdown/. Accessed: 2018-03-08.

[14] Facebook. https://www.facebook.com/. Accessed: 2018-12-08.

[15] Facebook Open Source. https://opensource.fb.com/#frontend. Accessed: 2018-12-08.

[16] file.io - Ephemeral file sharing. https://www.file.io/. Accessed: 2018-12-08.

[17] Games Made With Unity. https://unity.com/madewith. Accessed: 2018-01-08.

[18] Haskell Language. https://www.haskell.org/. Accessed: 2018-01-08.

[19] Holiday Rentals, Homes, Experiences & Places - Airbnb. https://www.airbnb.co.uk/. Accessed: 2018-12-08.

[20] Instagram. https://www.instagram.com/. Accessed: 2018-12-08.

[21] Introduction and Goals - Unity. https://unity3d.com/learn/tutorials/topics/scripting/introduction-and-goals. Accessed: 2018-01-08.

[22] javascript - How to make multiple chunk of lines readonly in ace editor - Stack Overflow. https://stackoverflow.com/questions/39640328/how-to-make-multiple-chunk-of-lines-readonly-in-ace-editor. Accessed: 2018-12-08.

[23] lens: Lenses, Folds and Traversals. http://hackage.haskell.org/package/lens. Accessed: 2018-15-08.

[24] Lifeline game. https://www.bigfishgames.com/daily/3mingames/lifeline/. Accessed: 2018-19-04.

[25] Lifeline: Whiteout - Apps on Google Play. https://play.google.com/store/apps/details?id=com.threeminutegames.lifelinewhiteoutgoog&hl=en_GB. Accessed: 2018-12-08.

[26] Lorem Ipsum - All the facts - Lipsum generator. https://www.lipsum.com/. Accessed: 2018-01-08.

[27] MarkLight | Unity Presentation Framework. https://web.archive.org/web/20180520225541/http://www.marklightforunity.com/. Accessed: 2018-01-08.

[28] MarkLight: Unity Presentation Framework - Asset Store. https://assetstore.unity.com/packages/tools/gui/marklight-unity-presentation-framework-37466. Accessed: 2018-01-08.

[29] Node.js. https://nodejs.org/en/. Accessed: 2018-01-08.

[30] QuickCheck: Automatic testing of Haskell programs. http://hackage.haskell.org/package/QuickCheck. Accessed: 2018-07-08.

[31] React - A JavaScript library for building user interfaces. https://reactjs.org/. Accessed: 2018-31-07.

[32] react-ace - npm. https://www.npmjs.com/package/react-ace. Accessed: 2018-05-08.

[33] Scalaquest. http://scalaquest.com/. Accessed: 2018-07-08.

[34] ScalaQuest - a game to learn Scala by Alejandro. https://www.kickstarter.com/projects/andanthor/scalaquest-a-game-to-learn-scala. Accessed: 2018-08-04.

[35] Screeps - MMO strategy sandbox game for programmers. https://screeps.com/. Accessed: 2018-17-08.

[36] Screeps: The javaScript Programmer's MMO. https://www.gameskinny.com/mv7sb/screeps-the-javascript-programmers-mmo. Accessed: 2018-17-08.

[37] Sites Using React. https://github.com/facebook/react/wiki/Sites-Using-React. Accessed: 2018-12-08.

[38] State and Lifecycle - React. https://reactjs.org/docs/state-and-lifecycle.html. Accessed: 2018-03-08.

[39] th-alpha: Alpha equivalence for TH Exp. http://hackage.haskell.org/package/th-alpha. Accessed: 2018-07-08.

[40] The RTS That Makes You Program Your Own Units. https://www.kotaku.com.au/2016/08/the-rts-that-makes-you-program-your-own-units/. Accessed: 2018-17-08.

[41] Unity. https://unity3d.com/. Accessed: 2018-31-07.

[42] Unity - Manual: Creating and Using Scripts. https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html. Accessed: 2018-01-08.

[43] Unity - Manual: UI. https://docs.unity3d.com/Manual/UISystem.html. Accessed: 2018-01-08.

[44] Unity for 2D | Unity. https://unity.com/solutions/2d. Accessed: 2018-01-08.

[45] User Mr. A - Stack Overflow. https://stackoverflow.com/users/3026829/mr-a. Accessed: 2018-12-08.

[46] Variant – Triseum. https://triseum.com/variant-limits/. Accessed: 2018-15-08.

[47] Variant Post-gameplay Results. https://s3-us-west-2.amazonaws.com/triseum-variant/collateral/Variant_Post-Gameplay-Results_SPR17.pdf. Accessed: 2018-17-08.

[48] write-file-atomic - npm. https://www.npmjs.com/package/write-file-atomic. Accessed: 2018-06-07.

[49] ECMAScript 2018 language specification. Standard ECMA-262, Ecma International, Geneva, CH-1204, June 2018.

[50] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, Cambridge, MA, USA, 1996, ch. 1.

[51] BRAY, T. The javascript object notation (json) data interchange format. Tech. rep., 2017.

[52] CLAESSEN, K., AND HUGHES, J. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not. 35*, 9 (Sept. 2000), 268–279.

[53] DONDLINGER, M. J. Educational video game design: A review of the literature.

[54] FU, R. Haskellquest: a game for teaching functional programming in haskell. Informatics Project Proposal, 2018.

[55] HUDAK, P., HUGHES, J., PEYTON JONES, S., AND WADLER, P. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), HOPL III, ACM, pp. 12–1–12–55.

[56] LIPOVACA, M. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[57] LIPOVACA, M. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011, ch. 1.

[58] MARLOW, S., ET AL. Haskell 2010 language report.

[59] MARLOW, S., AND PEYTON JONES, S. *The Glasgow Haskell Compiler*. Lulu, January 2012.

[60] MICHAEL, D. R., AND CHEN, S. L. *Serious games: Games that educate, train, and inform*. Muska & Lipman/Premier-Trade, 2005.

[61] NELSON, G. A short history of interactive fiction. *The Inform DesignerâĂŹs Manual* (2001).

[62] ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. An overview of the scala programming language. Tech. rep., 2004.

[63] O'SULLIVAN, B., GOERZEN, J., AND STEWART, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[64] O'SULLIVAN, B., GOERZEN, J., AND STEWART, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008, ch. 2.

[65] PARNAS, D. L., SHORE, J. E., AND WEISS, D. Abstract types defined as classes of variables. *SIGPLAN Not. 11*, SI (Mar. 1976), 149–154.

[66] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 23, p. 340.

[67] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 8, p. 92.

[68] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 5, pp. 51–65.

[69] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 9, pp. 99–103.

[70] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 23, pp. 341–350.

[71] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 5, pp. 60–61.

[72] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 5, p. 56.

[73] PIERCE, B. C. *Types and Programming Languages*, 1st ed. The MIT Press, 2002, ch. 9, pp. 106–107.

[74] ROUSE, R. *Game Design Theory and Practice*, 2nd ed. Wordware Publishing Inc., Plano, TX, USA, 2000, ch. 1, pp. 2–8.

[75] RUKAVICKA, J. Dickson's method for generating pythagorean triples revisited. *European Journal of Pure & Applied Mathematics 6*, 3 (2013).

[76] SHEARD, T., AND PEYTON JONES, S. Template meta-programming for haskell. pp. 1–16.

[77] SØNDERGAARD, H., AND SESTOFT, P. Referential transparency, definiteness and unfoldability. *Acta Informatica 27*, 6 (1990), 505–517.

[78] SORENSEN, E., AND MIKAILESC, M. Model-view-viewmodel (mvvm) design pattern using windows presentation foundation (wpf) technology. *MegaByte Journal 9*, 4 (2010), 1–19.

[79] STAFF, C. React: Facebook's functional turn on writing javascript. *Communications of the ACM 59*, 12 (2016), 56–62.

[80] SULZMANN, M., CHAKRAVARTY, M., PEYTON JONES, S., AND DONNELLY, K. System f with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)* (January 2007), ACM, pp. 53–66.

[81] SUSI, T., JOHANNESSON, M., AND BACKLUND, P. Serious games: An overview, 2007.

[82] THOMPSON, S. *Haskell: The Craft of Functional Programming*, 3rd ed. Addison-Wesley Publishing Company, USA, 2011.

[83] THOMPSON, S. *Haskell: The Craft of Functional Programming*, 3rd ed. Addison-Wesley Publishing Company, USA, 2011, ch. 1, p. 3.

[84] WASIK, S., ANTCZAK, M., BADURA, J., LASKOWSKI, A., AND STERNAL, T. A survey on online judge systems and their applications. *ACM Comput. Surv. 51*, 1 (Jan. 2018), 3:1–3:34.