University of Dublin

Trinity College

Quicksilver

A Processing like domain specific language in Haskell

Aidan Duggan

B.A.I. Engineering

Final Year Project: April 2010

Supervisor: Glenn Strong

School of Computer Science and Statistics
OReilly Institute, Trinity College, Dublin 2, Ireland

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercie for a degree at this or any other university.

Name

Date

Abstract

Non-strict functional programming languages provide a powerful and alternate way to develop software programs. However their use has been over shadowed by imperative languages. This is often reinforced by the fact that most programmers first experience programming through imperative languages. A recent success in introducing people to programming is Processing, "a programming language and integrated development environment (IDE) built for the electronic arts and visual design communities". Processing focuses on the key area of graphics, it encourages people to program by making it easy to produce simple visual programs that allow the coder to see what they have written in action.

It is therefore clear that an effective way to introduce people to a programming language is to provide an easy tool for achieving quality visuals. This gives the programmer immediate satisfaction and provides results that are appealing to the general public, while teaching the basics of programming. Quicksilver is such a tool, written in the functional programming language Haskell. The tool provides the user with the basic tools to create visuals, requiring the minimum of experience. Through this the hope is to slowly introduce them to functional programming. Quicksilver is made of three parts; a library of functions to create visual effects, a framework within which the user can program, and an IDE to provide the user with the simplest experience.

Acknowledgment

I would like to thank Glenn Stong my supervisor. He was always there to discuss problems and explain things no matter how big or small. He has been an endless source of help and guidance.

I would also like to thank my girlfriend Lisa, her endless encouragement and support insured not only that this project was completed but that I remained sane throughout the process.

I'd like to thank my parents for their support, particularly throughout my years in college, without them I would not have gotten this far.

And special thanks must be given to my cat Sox who's company was invaluable during the long days working on this project.

”If I have coded further then others it is because I have stood on the code base of giants.”

Issac Newton/Aidan Duggan

Contents

Section 1

Introduction

1.1 Purpose

The purpose of this project is to create a domain specific language and environment that makes it easy to produce graphics in Haskell, I have named it Quicksilver (Haskell Graphics - Hg - Hydrargyrum - Quicksilver). Graphics and animation are the most easily appreciated product of programming, everyone can find an interactive game enjoyable or see the beauty in a well produced animation. By making it easy for the user to start producing graphics we are giving them instant feedback that they can appreciate. This makes learning a language much easier and more enjoyable.

A language exists already called Processing which has similar aims, it is implemented in Java and provides an API to produce professional quality graphics. However Haskell presents a number of interesting features which makes this project a lot more challenging. Haskell is a functional language unlike the imperative Java Processing uses, and importantly Haskell is lazy making IO actions such as graphics more complicated. This can make learning Haskell difficult.

Quicksilver aims to provide an API and environment to make it easy to learn Haskell. With this aim there are two goals. First Quicksilver should be easy to use, the measure for this will be Processing. Second Quicksilver does not try to hide the complex aspects of Haskell, there is no point removing the parts that make Haskell worth learning from a tool which is teaching it. Processing teaches programming through Java and graphics, Quicksilver will teach functional programming through Haskell and graphics. Java's syntax and style will be replaced with Haskell syntax and style.

1.2 Layout

This project is comprised of four key chapters. Chapter 2 is the technical background, this will give a brief overview of Processing and an introduction to Haskell for those who have no experience with it. Chapter 3 discusses the design of Quicksilver, its API, architecture and IDE. Chapter 4 is the implementation, it highlights the key areas in Quicksilver's development and a number of challenges faced. Finally chapter 5 is a series of demonstrations of Quicksilver, going from the very basic commands to more complex functional programming style. It serves as an example of how Quicksilver can be used to teach functional programming and the type of graphics and animations that it can produce. This report is rounded off with a discussion of the work completed and conclusions, particularly an overview of areas where improvements could be made and initial design choices which could be reconsidered.

Section 2

Technical Background

This chapter introduces the fundamental technologies used in the development of Quicksilver. Much of the discussion of how these technologies affect Quicksilver has been placed in the appropriate future section.

2.1 Processing

2.1.1 Background

Processing[8] came out of a project called Design By Number in MIT Media Lab [5]. DBN aimed to teach programming through simple graphics, lines, dots and greyscale, and providing the simplest environment in which to program. While Ben Fry and Casey Reas were working on DBN, they experimented with expanding its graphical support, but found this did not fit the aims of DBN. They therefore founded Processing, taking much of the aim of DBN but expanding its graphical abilities aiming to provide both a more powerful tool to graphic designers and provide a stepping stone into other programming languages[7]. A large community has grown around Processing providing library's which vastly increase it capability. It would be safe to say that processing has and continues to achieve its aim of introducing programming to a wider audience.

2.1.2 Implementation

Processing language is basically Java with a new graphics API. When they were choosing what language to build Processing on, there were three key aspects they were interested in, speed, appropriate as an introductory language, and act as a stepping stone to more powerful languages. Java fitted these goals perfectly, faster than scripting languages but more forgiving then C++. The C like syntax also helps when moving to more powerful C like languages. Supporting it and what makes Processing simple to use is its IDE. When the user runs their code the IDE preprocesses it into Java code with the attached Processing library. In a sense the IDE hides all of the over head involved in setting up the code to run and to produce a window. When designing simple programs one would often use the same setup code, so it is quite sensible to hide it from beginners. Listing 2.1 and figure 2.1 shows an example graphic generated in Processing. Processing and how it compares to Haskell will be discussed further in following chapters, particularly highlighting the key areas of difference.

Listing 2.1: Sample Processing code

```
void setup()
{
  size(200,200);
}

void draw()
{
  rectMode(CENTER);
  rect(100,100,20,100);
  ellipse(100,70,60,60);
  ellipse(81,70,16,32);
  ellipse(119,70,16,32);
  line(90,150,80,160);
  line(110,150,120,160);
}
```

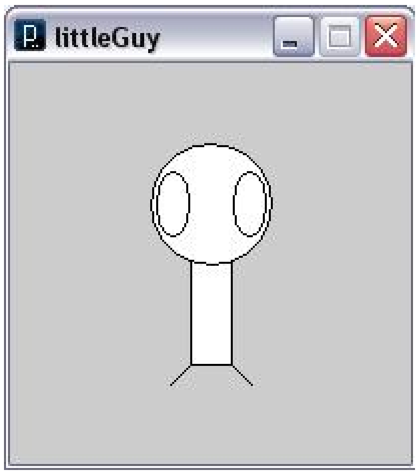


Figure 2.1: Graphic produced by code in listing 2.1

2.2 Haskell

2.2.1 Background

During the late 80's there was a lot of research into lazy functional languages. Dozens of such languages began to appear in academia. Functional and laziness will be discussed in section 2.2.2 and section 2.2.4 respectively. The most widely used was Miranda but it never reached the public domain. At the FPCA (Conference on Functional Programming Languages and Computer Architecture) in 1987 it was decided to create an open standard for a lazy functional programming language. The aim being that this language would be the basis of all further research in the area of functional language design. This new language was named Haskell[6].

Haskell was later refined into Haskell 98 which specified a stable, minimal, portable version with a standard set of libraries. The most mature compiler in present use is the Glasgow Haskell Compiler. The GHC was initially prototyped by Kevin Hammond at the University of Glasgow in 1989. It was later written in Haskell itself by Cordelia Hall, Will Partain, and Simon Peyton Jones and released in 1991. There exist many other compilers which specialize in exposing different aspects of Haskell, some aim to make Haskell easier to learn, Hugs, while others have been developed to do further research into functional programming, UHC. GHC is one of the few compilers' that can actually produce faster executing code then imperative language compilers. It is well supported and provides a fully implemented version of Haskell 98. For these reasons Quicksilver was developed using GHC.

2.2.2 Functional and Pure

The more popular languages like Java, C++ and C# are imperative programming languages. In such a language the programmer manipulates the state of the program which changes how parts of the program execute. The programmer performs actions which change the state of the program and through this produce the desired result. Functional programming takes a different approach, instead of performing actions on the state, functional programming involves declaring what values need to be computed and leaving it to the compiler and run-time to determine what order these values are computed in. Functional programming style makes extensive use of lists and recursion.

For a functional language (or any language) to be pure it must be referentially transparent. A function is said to be referentially transparent if replacing it with its value will produce the same result throughout the program. For example square(5)can be replaced with 25 and this would not effect the result, however if x were a C variable and we tried square(x) we cannot know what value x for all time and therefore cannot say square(x) will be referentially transparent, its value can change from one point to the next. Pure languages enforce referential transparency by enforcing immutable variables, once declared the value of a variable cannot be changed and so a function that takes the same inputs will always produce the same output since no value can have changed the second time. Along with this a function is prevented from causing side effects, for example accessing an external file. A good example of how this compares with imperative programming is discussed in section 4.2.4 along with how it directly affects the implementation of Quicksilver.

Some functional programming languages do not strictly enforce immutable variables and do not attempt to achieve referential transparency. In this way Haskell is significantly different, by enforcing referential transparency it allows for better reasoning about the correctness of a program. Where two equations can be brought to equality they are free to be interchanged. This makes it easier for the programmer to see what a program is doing and easier for the compiler and run time to determine better if optimizations can be achieved and spot errors of logic. The price for this is paid in larger memory space, also slower execution times can result. Like all such performance enhancers in some cases things can be faster and in others slower. The real power of referential transparency is the freedom it gives to reason about the program, something which referentially opaque languages do not support.

The key point of note is that functions cannot have side effects, the only affect they have on the program is the value they return. They cannot change the value of variables they can only produce new ones. In truth there are no real variables, only functions which return constant values. For the ease of distinguishing a function whose purpose is computing something and a function is which is acting as a value, the term variable is used throughout this report.

Haskell also makes use of Higher-order functions. Higher-order function either take a function as an input or return a function as an output. A good example is map, map has the Haskell signature map :: (a -> b) -> [a] -> [b]. It takes a function (which takes a value and returns a value) and list of the same type the function takes. It then maps that function across all the entries in the list producing a new list which map outputs.

2.2.3 Syntax

What follows is a few key areas of Haskell syntax so that future code snippets can be better understood.

Lists

Here is some of the syntax for creating lists. Lists are an important data structure in Haskell, a number of predefined functions are provided to support the manipulation of lists.

Listing 2.2: Creating lists in Haskell

```
--A function which returns a list
newList = [1,2,3,4]

--Adding a value to a list: newerList will be [1,1,2,3,4]
newerList = 1:newList

--Adding two lists: newestList will be [1,1,2,3,4,1,2,3,4]
newestList = newerList++newlist

--Get the first entry in a list: entry will have the value 1
entry = head newList

--Get all but the first entry: restOflst will be [2,3,4]
restOflst = tail newList

--Each name is a function definition.
newList = 1:newerList
{-This will result in a compiler error
since newList is already defined above.-}
```

Data types

Syntax to create tuples and records. These are useful for storing groups of data types which are in some way related.

Listing 2.3: Creating tuples and records in Haskell

```
-- Tuples are pairs of types,
--can also be triples, quadruples and so on.
newTuple = (1,'a')
newTriple = (1,'a',"first")
```

```
--Any type can be used in a tuple,
--easy access is provided through fst and snd
--firstEntry is the Integer 1, secondEntry is the string "One"
firstEntry = fst (1,"One")
secondEntry = snd (1,"One")

--Accessing and creating tuples/triple etc can be
--cumbersome. Records provide an easier
--tool for frequently used data structures
data NewRec = { num :: Int, letter::Char, word ::String}

--Records are created as if the name is a function,
--which in a sense it is.
createNewRec = NewRec 1 'a' "first"

--Values are accessed using their names,
--passing the relevent record as a parameter.
--recInt will have the value 1
recInt = num createNewRec

--New records with similar values can be created easily.
--newerRec will have the values 1 'a' and "One"
newerRec = createNewRec {word = "One"}
```

Functions

Syntax for defining functions, including pattern matching.

Listing 2.4: Declaring functions

```
--This function takes two Integers and adds them.
addInt :: Int -> Int -> Int
addInt x y = x + y

--This function puts two things into a tuple.
--These can be of the same or different type.
pairUp :: a -> b -> (a,b)
pairUp x y = (x,y)

--pattern-matching allows values to be tested
--at the boundary of the function call.
--They do not have to test everything
--but must handle the general case.
multiple :: Int -> Int -> Int
multiple 0 x = 0
multiple x 0 = 0
multiple x y = x*y

--Iteration is achieved through
--recursion and pattern-matching.
--Here we add a number to each entry in a list.
--Lists can be matched to top and rest.
--Pattern matches are found from top down.
addValueToList :: Int -> [Int] -> [Int]
addValueToList y [] = []
addValueToList y (x:xs) = [x+y]:(addValueToList y xs)
```

2.2.4 Lazy

Giving the compiler and run-time the ability to make better optimizations is advantageous, so Haskell goes a step further by implementing lazy evaluation. This and higher-order function has been argued to be the glue for powerful functional programming[3]. Laziness allows the run-time to put off calculations until/if the result is needed. So if a function would calculate the square of its parameter, the value would not actually be calculated until the result is needed, for example when it is printed to the screen or checked in an if statement, see listing 2.5 for examples. Until that point the program holds what is called a thunk which references what calculation needs to be performed to achieve the actual value. This means if a value is never used it will never be calculated.

This allows for functions to have infinite results since only those used will be calculated. For example imagine we have a function that returns an array of all the primes, in a strict language the programmer would have to specify when the array should end or the function would never terminate. However in Haskell the function does not calculate anything till the value is needed and it remembers the values it previously calculated since its output cannot change if the input is the same. The price that is paid for this is in memory, thunks can be quite large and for small calculations outweigh the benefits. The programmer can force evaluation using a keyword 'seq' but it can be hard to reason where it is most efficient to do so.

Listing 2.5: Examples of Lazy evaluation

```
--Here is our square function
square x = x*x

--Here is a list of squares
listOfSq = [square 2, square 3, square 4]

--If we were to examine the list more closely
--it would actually look like this:
listOfSq = [2*2, 3*3, 4*4]

--Only if the value is needed will it be calculated
if((head listOfSq) > 5)
        then
        --do something
        else
        --do something else

--Now the list will look like so:
listOfSq = [4, 3*3, 4*4]

--Laziness allows for infinite lists
--Only values needed are computed
infiniteSeries x = x : infiniteSeries (x+1)

--We can start the series with a value
```

```
createSeries = infiniteSeries 2

--It will look like this:
createSeries = 2:infiniteSeries (2+1)

--After accessing the first few elements
--createSeries will look something like this:
createSeries = [2,3,4,5] ++ infiniteSeries (5+1)
```

2.2.5 Monads

So now we have a pure and lazy language, a function with the same input must return the same output and the order in which calculations are preformed is no longer up to us. It all sounds good but what two main aspects of programming might this cause trouble for? Reading inputs from the key? The function to return a key press will always take the same parameters (nothing) but return different output, purity does not work here. What about printing messages to the screen? What if we print "Hello" and "World" to the terminal which will appear first? Laziness or more particularly non-strict evaluation does not work here.

It is therefore clear we need some structure to handle IO but there is a more general case here. We need a tool for when the order of actions must be controlled and when a series of actions are interacting with a common piece of data. In IO we need to be sure read and write order is correct and all actions need access to information about the IO back-end. Another example might be a text parser, all actions need to happen in order, and they all need access to the text being parsed. The final thing that is often needed is some way to get information into this environment, to load the text that is being parsed or some computation whose results need to be printed to the screen.

The solution to this problem was Monads[9]. A Monad is an abstract data type which provides for the ability to chain actions together. Monads are defined by a Kleisli triple: a type construction that defines how to create the monad from a data type, this is the name of the monad in Haskell, a unit function which lifts a data type into the monad, called return in Haskell, and a binding operation which chains two monad actions together, called >>= in Haskell.

So you can see this structure fits perfectly with what we need to do IO. To achieve IO the bind action is defined to perform the non-pure IO action and ensure the correct order of execution. In this way we avoid the issue of laziness and purity without compromising the language. These impure actions are restricted to the IO monad and it is explicit to the programmer and the compiler what might go on. In a parser bind would pass along the parsed text and the text to be parsed. In the case of the parser nothing untoward is happening, monads simply make passing state much similar and cleaner, it is still pure.

This passing and managing state is actually particularly important. IO is by definition hard to handle in a pure and lazy language so exceptions can be made, but managing state is integral to functional programming. Monads abstract away the tricky part of passing around a value between each function call and gives the impression of having a mutable store while achieving it in a pure and safe way. Monads still require the programmer to define and implement the 3 features above and so require a lot of work. Much of this project revolves around integrating IO and state management and handling this problem.

Monads are implemented in other programming languages but Haskell provides the simple do-notation and the power of monads is more apparent in functional languages. Listing 2.6 shows an example with the actual monad notation and the do-notation which is a syntactic sugar. The (_->) is called an anonymous function, it allows a function to be defined in-line and applied immediately, the _means it does not matter what parameters it is passed. If you look closely at the actual notation you will see that each line is bound to an anonymous function _-> which executes all the following lines, with each line equally bound to a function of the following lines. When a variable is defined using <- it is passed as a parameter \name instead of _. You can see that monads are quite complicated to work with, without the do notation. More features of managing state in Haskell and how it compares to imperative languages will be discussed later in section 4.2.

Listing 2.6: Example of Monads

```
--This is the do notation:
do
        putStrLn "What is your name?"
        name <- getLine
        putStrLn ("Nice to meet you, " ++ name ++ "!")

--This is the actual notation:
putStrLn "What is your name?" >>=
   (\_ ->
      getLine >>=
         (\name ->
            putStrLn ("Nice to meet you, " ++ name ++ "!")))
```

2.3 wxHaskell

2.3.1 Background

wxHaskell was designed to be a portable and concise GUI library for Haskell [4]. One of the key aims of wxHaskell is to create a platform independent library that still provides the look and feel of the system it is running on. wxHaskell produces fast and native looking applications with a wide range of graphics and UI. It is a standard go to library for building GUI's in Haskell and by supporting a wide range of platforms made it to good library upon which to build Quicksilver.

2.3.2 Implementation

Realising the amount of time it would take to write their own graphics, the developers of wxHaskell looked for an already established graphics library on which to build. They choose wxWidget, it is a C++ library that provides a common interface to UI widgets on all the major OS's, and it has a strong development community and is in wide use. wxHaskell is made of two main libraries, WX and WXCore. WXCore was built on top of wxWidget.

The wxWidget function calls were given a C wrapper and then called through Haskells foreign function interface. This provided the necessary tools for producing graphical widgets. WX was built on top of WXCore adding abstraction and allowing it to us Haskells type class overloading, higher-order functions, and polymorphism to make wxHaskell easier to use. wxHaskell uses mutable stores to pass values around, at the time there was no fully developed way to do a purely functional GUI. Many people have gone on to use wxHaskell as the base for such a library, where everything is purely functional. Listing 2.7 and figure 2.2 are a sample wxHaskell program, creating a simple window with a quit button.

Listing 2.7: Example of a wxHaskell program

```
--Produces a window with a quit button
main :: IO ()
main
  = start hello

hello :: IO ()
hello
  = do f    <- frame      [text := "Hello!"]
       quit <- button f [text := "Quit", on command := close f]
       set f [layout := widget quit]
```

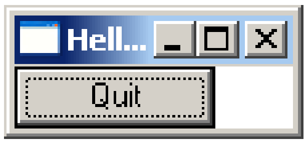


Figure 2.2: Graphic produced by code in listing 2.7

Section 3

Design

This chapter gives an overview of how Quicksilver was designed. Discussing Domain specific languages, giving an example of Quicksilver and laying out it architecture. Then discussing the design of Quicksilvers API and supporting IDE.

3.1 Domain Specific Language Structure

3.1.1 Quicksilver Overview

The more a programming language is tailored for its purpose, its domain, the more powerful it is within that domain and the quicker this power can be brought to bear. This is achieved by ensuring the language has the tools needed to complete the task. Also by designing the language for the domain, it can be created so resulting code clearly expresses how a domain problem is being solved. This makes it both easier to write and read these domain specific programs.

Since the aim of Quicksilver is to introduce people to functional programming, and we have seen that a good method for this is graphics then clearly a domain aimed at producing functional graphics and animation would be best for the task. The problem however is that, creating a language from scratch is not exactly easy, there is a mountain of work from syntax to compiler and that is before any domain specializing. The solution is to use an existing language and create a domain specific embedded language. Using this existing language as a foundation around which to build the domain language. Haskell is a clear choice for this[2].

So let us take a quick overview of Quicksilver before getting into its architecture and implementation. To do that we will go through a simple example program, a circle which changes colour. The user must provide implementation for four functions, the UserRecord, the defaultUserRecord, the setup and the painting function. These functions will all be automatically called by Quicksilver to produce the graphics and animation. The setup and painting function both hand the user a UserRecord and expect the user to return a UserRecord. The use of return is actually the monadic sense (lifting UserRecord into the Quick Monad) but the result is the same, since the value of the last action performed will be the output of the painting function. The user is free to replace this with any action whose output type is Quick UserRecord. These functions will all be discussed in more detail in future sections but for now here is a brief breakdown. Listing 3.1 shows the empty implementation of these four functions.

Listing 3.1: Empty implementation of a Quicksilver program

```haskell
data UserRecord = UserRecord{
        screenX :: Int,
        screenY :: Int
}
defaultUserRecord :: UserRecord
defaultUserRecord = UserRecord 200 200

setup :: UserRecord -> Quick UserRecord
setup record = do
                --Do Setup Here
                return record

painting :: DC a -> UserRecord -> Quick UserRecord
painting dc record = do
                -- Generate Graphics Here
                return record
```

UserRecord

The UserRecord is a Haskell Record data type. This is where the user can declare any variables they wish to use. They must provide for the screen dimensions.

defaultUserRecord

defaultUserRecord function is used to set the initial values for all variables defined in UserRecord. It is normal Haskell and so limited to non-monadic actions.

setup

setup function is used to do any once-off computations or initializations before the graphics are generated. It is in the Quicksilver Monad and so monadic action are allowed, e.g. random numbers to seed the graphics.

painting

painting function is used to generate graphics, it is repeatedly called and is where the user draws their shapes.

First we need a variable to store the colour of our circle, and we need to initialize it with some values. To generate the colours well create a list of triples which loops through from (0, 0, 0) to (255, 255, 255) repeatedly. This is where laziness allows us to create an infinite repeating list of colours. We create this repeating loop using Haskells pattern matching and just build up a list of triples.

Listing 3.2: Setting up the colour changing circle

```
data UserRecord = UserRecord{
        screenX :: Int,
        screenY :: Int,
        cols1  :: [(Int,Int,Int)]
}

defaultUserRecord = UserRecord 200 200 (genColour 0 0 1)

genColour 255 255 255 = (255,255,255):genColour 0 0 1
genColour r 255 255 = (r, 255, 255):genColour (r+1) 255 255
genColour 0 g 255 = (0, g, 255):genColour 0 (g+1) 255
genColour 0 0 b = (0,0,b):genColour 0 0 (b+1)
genColour r g b = (r,g,b):genColour r g b
```

Before we draw our circle we use the setup function to set our background to white. setBackgroundColour does as its says, taking the given triple and setting the background colour to the red green blue value. Now we use painting to draw our circle, first get our colour, the record variable is our UserRecord and from it we extract our cols1 list. We take the head of the list and pass it to setFillColour. This determines the colour which all future shapes will be filled with. We then draw our circle in the centre. Finally we need to update our list since we have used the first entry. To do this we return a new record which is the tail of our current list. Now with each loop the top of the list will be used and then chopped off.

Listing 3.3: Drawing the colour changing circle

```
setup record = do
               setBackgroundColour (255,255,255)
               return record

painting dc record = do
               setFillColour (head (cols1 record))
               circlePro dc (100,100) 50
               return record {cols1 = tail (cols1 record)}
```

This is just a brief example to give a feel for how Quicksilver is used and prepare you for the next few chapters, more detailed examples will follow in chapter 5.

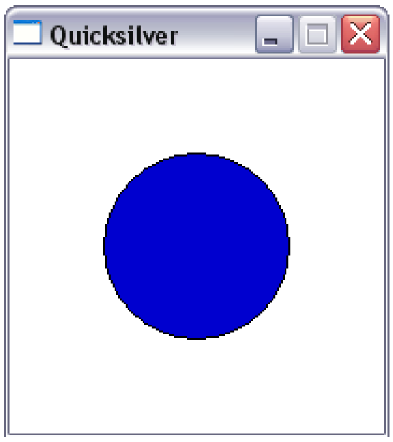


Figure 3.1: Image produced by code in listing 3.2 and 3.3

3.1.2 Quicksilver Architecture

Quicksilver can be thought of as a layered architecture, with wxHaskell forming the base. wxHaskell provides the graphical abilities and the storage of state between frame calls. On top of that is the State Management which manages the variables used in generating the graphics. On top of this is the new graphics interface, which provides the user with the ability to draw shapes, changes colours, etc. The next layer is the user code which uses the functions provided by the graphics layer. The final layer is the program layer, this ties together the lower layers to produce the animation. Figure 3.2 shows the layout of the architecture.

wxHaskell Layer

wxHaskell provides the core support of Quicksilver. It provides all of the graphical functionality as described in section 3.2.2. It is used to create the program's window and start the setup and painting loop. It generates the call-backs and stores the state, passing it between each frame of the program. wxHaskell provided a number of the key features need for Quicksilver, the challenge lay in remodeling them to make it easier to use and hide the code overhead from the user.

State Layer

The State layer is used to manage the state of the users program. wx-Haskell is used to store the state. The state is a Haskell Record data type, which allows a number of variables be combined and more easily accessed and updated. The record carries information about the colour and styles the user wishes to use, along with any inputs that have been received such as key's pressed or position of the mouse. The UserRecord is stored in the larger record. This larger record is wrapped in a State monad which allows to state to be passed and used without direct coding from the user. An API is added to make the values stored in the record easily accessed.

Graphics Layer

The Graphics layer merges the two lower layers of wxHaskell and State. In order to pass the State around seamlessly it requires the use of the State monad, wxHaskell requires the use of the IO monad. Therefore a new monad had to be use, called the StateT monad. This allows State and IO to mixed and will be referred to as the StateIO monad. The Graphics layer takes the values that are present in the State and applies them to wxHaskell to produce shapes with the appropriate style. This is presented as an API of shapes and styles the user can access.

User Layer

This is where the user provides the four key functions, the UserRecord, defaultUserRecord, setup and painting. These functions can call on the Graphics and State layers below to achieve the graphics which the user wishes to generate.

Program Layer

The Program layer sets up the environment in which the user code runs, combining all the previous layers.

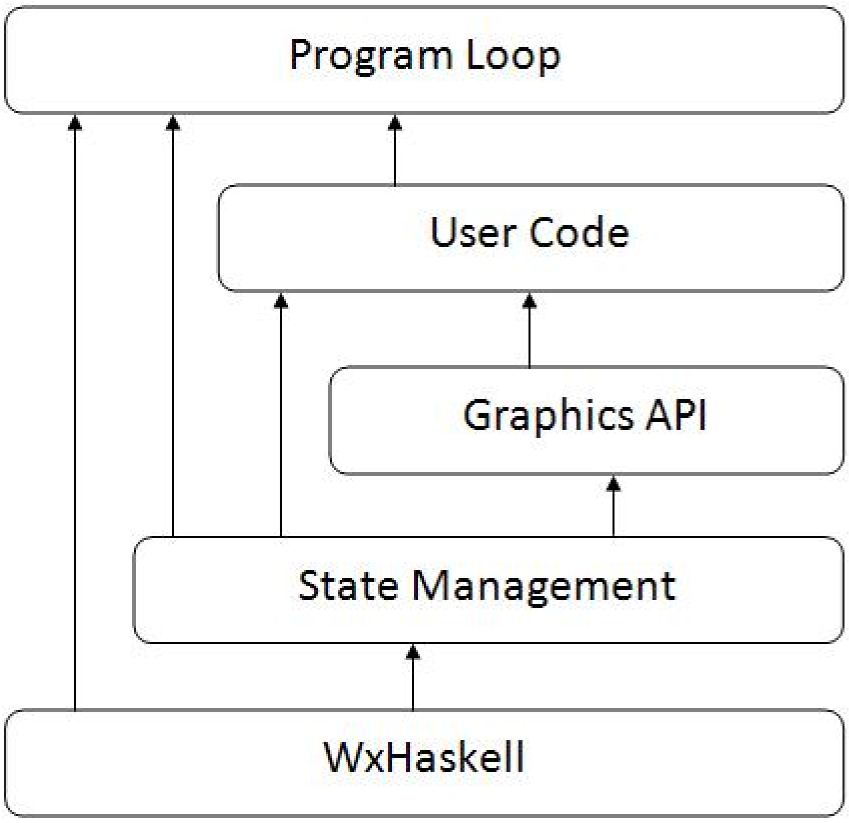


Figure 3.2: Architecture Diagram

3.2 API Decisions

3.2.1 Processing API

The Processing API can in a sense be divided into two main sections. The first is the keywords, types and general functions that are required to program in Java, e.g int, double, for, while, abs, sqrt etc. The second section is the library of functions for producing graphics. Processing currently has quite an extensive graphics set, ranging from 2d primitives to full blown opengl along with libraries for other forms of IO such as sound, video, network and much more.

Setting up a Processing program is achieved by providing the implementation for the function setup() and for animations the function draw() as well. The setup function is called at the start of the programs execution and is where things like screen size and initializing variables can be programmed. Draw() function is called repeatedly to produce the animation. In a way the user is overriding the base implementation of these functions with new functions. The advantage of this form of function overriding is that should the user not provide an implementation a default still exists. This is the way Processing implements call back function for inputs, if the user does not provide what should happen when a key is pressed for example, the default implementation will take care of it. These are all standard aspects of the Java language which are being leveraged to make Processing easy to use.

Graphics are controlled similar to OpenGL, declaring what style you want to use as its own statement with all follow graphics drawn with those styles. The alternative would be to specify the style when drawing each element but this would lead to excessive duplication. Processing allows the colour, transparency, drawing position and line thickness (as examples) to be set in this way. The Processing API is designed to achieve the stated aims of Processing, straight forward to use while introducing the user to programming and the concepts involved. It would have been possible to restrict the user to a smaller set of actions and so make it even simpler. However Processing aims to be a stepping stone to more complex programming and so to hide too much would defeat this purpose.

In Processing inputs are handled in two ways. Global variables can be accessed with information about the last key pressed, mouse button clicked and current mouse position. Call-backs can be attached to certain actions (mouse click, button pressed etc.) the user simply provides a function with the appropriate function signature. This works well in Java since the user is simply overriding a base class version, the program is not affected either way, since an implementation is always present.

3.2.2 Quicksilver API

One of the important aspects of Processing is it leaves Java untouched, it does not change the syntax for how Java works. This allows Processing to act as a stepping stone to full Java. This project is aiming to design a language similar but not identical to Processing. While it would be possible to make Quicksilver almost identical to Processing this would not achieve the aim of introducing the user to functional programming. This aspect was important when deciding on what to implement from the Processing API, the aim of the Quicksilver API is to keep as much of the Haskell syntax and style intact while mapping the way in which Processing implements graphics and design.

The challenge here is that Processing is an imperative language and Haskell functional. Decisions have to be made about where it is important to accept imperative style and where it is important to implement functional. As mentioned the functional style is important so where possible Quicksilver looks to encourage it. The problem is that graphics and more particularly IO are far easier to implement and use in an imperative style. For this reason Quicksilver copies Processings style of graphics API, where the user calls action which define the style that following shapes are drawn in. Beyond that however everything is left in a functional style, there is no attempt to provide a mutable store, users must manage their variables using the provided state management system which encourages programming functionally. In order to remain manageable the user must make us of tuples and lists which encourage the use of Haskell's predefined list manipulation functions.

The Quicksilver API is broken up into 5 parts, the the API for the last 4 parts is in appendix A. A program section ties the user code together, and is discussed in section 4.1.

Haskell
: Normal Haskell syntax and types are permitted, along with the standard library. This forms the base of programming in Quicksilver.

Graphics Types These functions are used to define what style shapes will be drawn in.

Graphics Shapes These functions are the range of 2D shapes that Quicksilver provides.

Input These functions allow the user to access any inputs received between frames.

Utility These functions provide the odd facilities that might be needed, particularly random numbers and printing to the terminal.

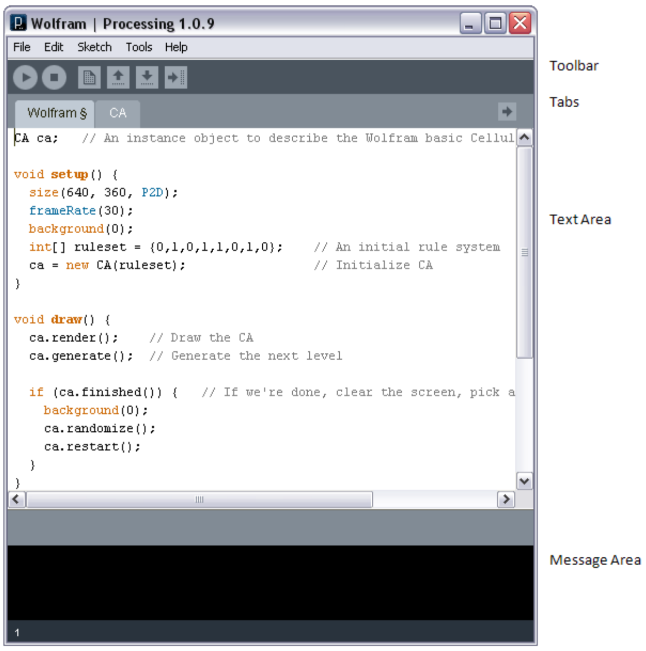
The scope of Processing API means it would be impossible to implement everything, therefore Quicksilver only implements a fraction of the API. Quicksilver provides for the basic 2d primitives: lines, rectangles and ellipses etc. It does not implement 3d or OpenGL. However because of Haskell's use of lists Quicksilver provides an different and intuitive implementation for polygons and polylines. Quicksilver provides the standard styles which Processing implements, allowing the user to dictate the fill colour, the line colour, line size and whether the object is drawn based on the centre point or top-left point. These work in the same way as in Processing, the user simply states the colour to be used and all following shapes are drawn with the colour. wxHaskell unfortunately does not provide the ability to set the transparency of shapes so this could not be supported in Quicksilver. Quicksilver allows for images to be used however this support is very rudimentary.

With regards to inputs Haskell unfortunately does not provide a mechanism for function overriding like in Java and so it is not possible to handle program input in that manner. A number of different possibilities present themselves which are discussed in section 4.2.3. The design goals for handling inputs was to present a simple and uniform way for the user to gain access to input information. The solution was to use standard API calls which access state that has been set between frames to what inputs have been received.

3.3 IDE Design

3.3.1 Processing IDE

The user is introduced to Processing through the Processing Development Environment (PDE), figure 3.3. This tool provides the user with everything they need to create a Processing animation. It can be separated into 3 key areas, the menu system, the editable area and the output section. The menu provides the load and save functionality, and also allows the code to be executed in one click. There are also options for better formatting the code and similar but these are ancillary. The editable area is where the user can write their code, it is tabbed to allow multiple classes to be opened on separate tabs. On starting this area is blank. The output section takes up the bottom quarter of the screen. When the code is executed any compiler errors are presented here. The PDE provides the user with a clear palette and makes developing and executing code easy. This is one of the reasons Processing is successful, by provide an IDE as part of the language it makes it much easier to hide a lot of the overhead involved in compiling programs.

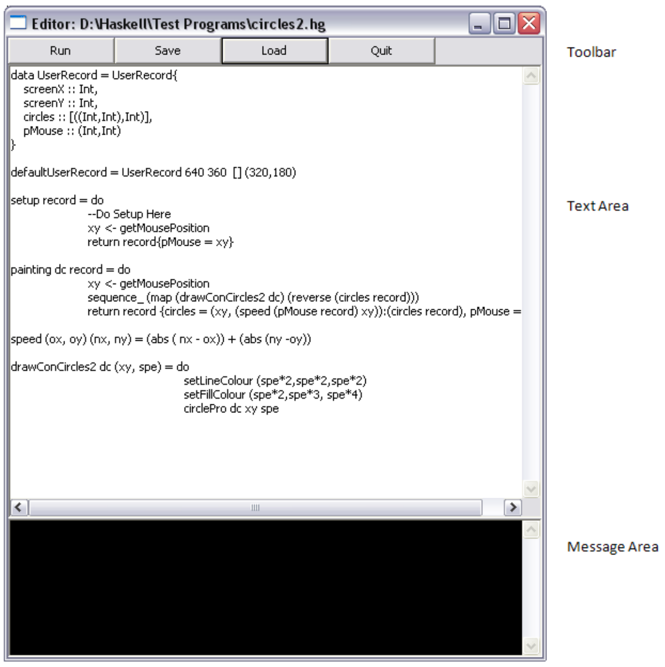


```
CA ca;   // An instance object to describe the Wolfram basic Cellul

void setup() {
  size(640, 360, P2D);
  frameRate(30);
  background(0);
  int[] ruleset = {0,1,0,1,1,0,1,0};    // An initial rule system
  ca = new CA(ruleset);                 // Initialize CA
}

void draw() {
  ca.render();    // Draw the CA
  ca.generate();  // Generate the next level

  if (ca.finished()) {   // If we're done, clear the screen, pick a
    background(0);
    ca.randomize();
    ca.restart();
  }
}
```

Figure 3.3: Processing Development Environment

3.3.2 Quicksilver IDE

Although not part of the project specification it was clear that Quicksilver would benefit greatly from having an associated IDE. An IDE would make it easier for the programmer to develop Quicksilver graphics and learn Haskell. This would better achieve the aim of introducing people to functional programming. The Quicksilver development environment, figure 3.4, was developed using wx-Haskell. The three areas highlighted in the Processing development environment were created. A simple menu to allow the load and saving of files and a button to execute the users code. A text area is created for the user to generate their graphics. This is loaded with the skeleton code in which the can insert their code. Processing does not provide their default code, since the skeleton code of Quicksilver is a little more complex, to make it easier on the user the QDE provides the default initial code. The final section is where the compilers output is displayed. The compiler is a command line whose input and output are fed through the QDE. The users code is compiled and executed showing the animation.



```
data UserRecord = UserRecord{
   screenX :: Int,
   screenY :: Int,
   circles :: [((Int,Int),Int)],
   pMouse :: (Int,Int)
}

defaultUserRecord = UserRecord 640 360  [] (320,180)

setup record = do
                  --Do Setup Here
                  xy <- getMousePosition
                  return record{pMouse = xy}

painting dc record = do
                  xy <- getMousePosition
                  sequence_ (map (drawConCircles2 dc) (reverse (circles record)))
                  return record {circles = (xy, (speed (pMouse record) xy)):(circles record), pMouse =

speed (ox, oy) (nx, ny) = (abs ( nx - ox)) + (abs (ny -oy))

drawConCircles2 dc (xy, spe) = do
                                          setLineColour (spe*2,spe*2,spe*2)
                                          setFillColour (spe*2,spe*3, spe*4)
                                          circlePro dc xy spe
```

Figure 3.4: Quicksilver Development Environment

Section 4

Implementation

This chapter will discuss the implementation of Quicksilver. It will focus mainly on a number of tough challenges faced during development to give a sense of how it was constructed. The challenges presented are each in key areas of development and in some cases directly shaped Quicksilver. To better understand their effects this chapter expands on some of the information presented in Chapter 2.

4.1 Program Execution

4.1.1 Program Loop

The key area which ties Quicksilver together is the program loop, this is the code which sets up the window and calls the users code to generate the graphics. The initial set up involves launching the window, initializing the State and storing it, attaching the call-backs to handle input and executing the users setup function. Then a timer is launched to repeatedly setup and call, the users painting function which generates the graphics. This loop takes a number of steps between start and finish. First the State is extracted out of wxHaskell's reference. The UserRecord is obtained from the state. The State is then used to run a StateIO Monad which executes the users painting function. The UserRecord is passed as a parameter to the painting function to make it easier for the user to use their variables since they will have the direct reference. The users code executes within the StateIO Monad and on completion they must return a userRecord (this could be new or not). When execution is completed the StateIO Monad returns the updated State. Some State variables are reset, the UserRecord is then attached and the result stored using wxHaskell for the next loop execution. This loop is displayed in figure 4.1.

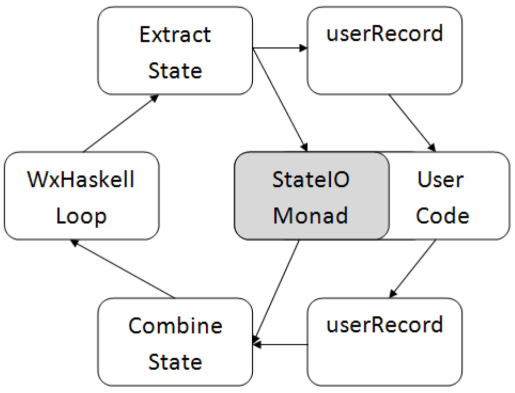


Figure 4.1: Program Loop

One of the challenges in developing this loop was managing the time between each loop. The timer which controls the rate at which the loop is repeated is stored in the State. When each loop starts the timer is stopped and just before the State is stored, ready for the next loop, the timer is started again. This way the time between loops depends on how long it takes the code to execute. A number of attempts were made before this solution was found.

At first timer was set to repeat every 60 milli-seconds. This took no account of how long the computation would take, which meant that problems occurred for long computations. If the loop computation took longer than 60 mill-seconds the loop immediately starts again. This would lead to threads of execution backlogging which resulted in the window not exiting correctly when the top-right x is clicked.

The next step was naturally to stop the timer during execution. The first attempt involved stopping and starting the timer between the call to repaint the window. The repaint function is part of wxHaskell and is a threaded event call, notifying the window that it should repaint its contents at the next opportunity. Therefore the start and stop had no effect, the time difference was only the time to notify the window not repaint it.

The solution then was restarting the timer just as the loop is finishing, however there is no direct way to pass the reference for the timer. So, as stated, the timer has to be added to the program State. The time was reduced to 30 milli-second to take into the count the time spent executing code. The key difference now is that the frame rate is dynamic, depending on how long the users code takes to execute. This is handy in that it gives feedback to the user if their code is particularly slow, however it can produce uneven frame rates which are not ideal for certain animations. Processing handles this in a similar manner except provides functionality to smooth the frame rate over time, this could have been implemented with sufficient time, but the aim was to produce a working project so the issue of stopping stacked event calls was priority over producing smooth frame rates.

4.2 State Management

4.2.1 Imperative versus Functional

The managing of state is one of the key differences between imperative and pure functional programming. In an imperative language state (variables) are created when and where they are needed. The location of their creation determines their scope and with it their likely use. Variables declared at the top are global and can be referenced and changed throughout the code. Their use might often be to allow functions to access a common variable without it being explicit. If every function needs access to the screen there is no need to pass it explicitly each and every time.

This has its problems, not only for the developer but particularly someone working with the code at a later date. In a sense every function is being passed every global variable and determining which are actually accessed would require actually looking in every function. Code can become very cluttered and interconnected in a way not obvious. The issue is not so much that these variables can be accessed, simply knowing the dimensions of the screen is not a problem. It is the ability to change these values, or for someone else to change them, that causes the problem. This can make it hard to ascertain where the variable is being changed or what problems changing it might cause.

Listing 4.1: Examples of global problems

```
int globVar = 1;

void printValue(){
        cout << globVar;
        globVar = 2;
}
int main(){
        globVar = 1;
        printValue();

        if (globVar == 1)
                cout << "A";
        else
                cout << "B";
        return 0;
}
```

Haskell prevents these issues but making functions pure and variables immutable. Since all variables are functions which take no parameters then their output can never change. This allows global access to constants such as screen dimensions but does not allow for global variables to be changed. The only way to achieve this affect is to explicitly return the new value, which would have to be passed to all further functions with an interest in using it.

Listing 4.1 will result un-expectingly in a B getting printed, listing 4.2 is an attempt to do the same in Haskell. Haskell results in a much safer version because there is no way to do the unsafe version presented above. In order to increment globVar printValue must return a new variable, in this case called newVar, in this way its clear to the programmer that this value might be something new and different, but they also know that globVar will still always be '1'. The printing of B is now not unexpected, we were aware of the risk that newVar might be different then globVar.

Listing 4.2: Lack of global problems in Haskell

```
globVar = 1

printvalue = do
                putStrLn globVar
                return (globVar+1)

main = do
        newVar <- printvalue globVar
        if(newVar == 1)
                then putStrLn "A"
                else putStrLn "B"
        return ()
}
```

Another use for variables is on the very local scale, perhaps a counter for a for/while loop or some other incremental store, listing 4.3 and listing 4.4 is an example in C and Haskell respectively. It is possible in Haskell to create something similar, a function which has to iterate a number of times is passed that number and it recursively calls itself decreasing the number till the number reaches a terminator. Alternatively if the number of times is not explicitly passed in but known, a new function can be created with the number passed in to be recursively called and reduced. This hides the variable use from the top level of the function.

Listing 4.3: Example of iteration in C++

```
int main(){
        int sum = 0;
        for(int i = 0; i < 50; i++)
        {
                sum = sum + i;
        }
        return 0;
}
```

Listing 4.4: Example of iteration in Haskell

```
sumFunction :: Int
sumFunction = sumFunction2 50

sumFunction2 :: Int -> Int
sumFunction2 50 = 50
sumFunction2 i = i + sumFunction (i+1)
}
```

It is clear that in Haskell variables do change but their change actually produces a new variable. This must be the result of a function which takes that variable as an input (except at the top where the variable is created) and returns the new version as an output. In imperative programming variables are stores for values which a function can change. It is clear managing state within Quicksilver will be considerably different from Processing. We will see in the following section how this conflict between managing variables is handled in Quicksilver.

4.2.2 Managing Graphics State

The way in which Processing implements Graphics generation is similar to OpenGL. Each statement is used to build up the graphics. For example setting what colour to draw in is its own statement, all future shapes are drawn with that colour until it is changed. Actions which change the manner and style shapes are drawn with are achieved in this manner. This way of implementing graphics shortens the length of function calls and spares repeated calls (such as setting the colour for each shape). wxHaskell on the other hand implements an API where graphics attributes are attached to the shape as a list.

Since Processing is the example of an easy to use API it was necessary to map wxHaskell to the Processing style API. This requires a state to be carried around controlling what colour and what style shapes are drawn in. Listing 4.5 and figure 4.2 shows an example of Processings API.

Listing 4.5: Examples of Processing API

```
void setup()
{
        size(200,200);
        stroke( 0,255,0); //The colour of the outline
        fill(0, 0, 255); //The colour to fill the shape in
        rect( 0 , 0, 100 , 100); //A shape
        ellipse(150,150,100,100); //Another shape
}
```

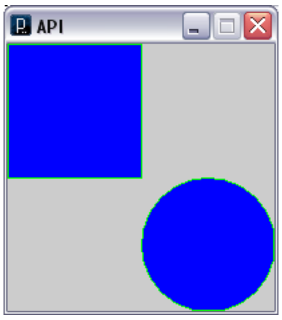


Figure 4.2: Image produced by listing 4.5

This graphics state has to be mutable and needs to be passed in a way that does not require extra effort on the programmer. Although these requirements go counter to Haskell's design they are often a requirement in development. Therefore Haskell provides the Monad structure as described in section 2.6, particularly the State Monad. This provides a syntactic sugar for sequentially passing a variable through a chain of functions. The passed variable is accessibly by any State Monad function and a function can create a new variable to be passed, as long as the type is the same. This may seem like cheating but Monads are an explicit structure and so when coding in a Monad the developer is aware that there is a hidden state and aware of what it stores.

The State Monad can only store one variable, however Haskell provides the Record structure, as described in section 2.2.2. Quicksilver uses this state to manage the styles of the graphics drawn and to pass inputs between frames to the user. This Record is a large number of variables, storing colours, styles, inputs etc. Listing 4.6 and figure 4.3 show an example of this state being passed without the it being explicit. setLineColour and setFillColour are actually creating a new state which is being passed to the next line.

Listing 4.6: Examples of Quicksilver API

```
painting dc record = do
          setLineColour (0, 255, 0) --The colour of line
          setFillColour (0, 0, 255) --The colour of fill
          rectPro dc (0 , 0) (100, 100) --A shape
          ellipsePro dc (100, 100) (100, 100) --Another shape
          return record
```

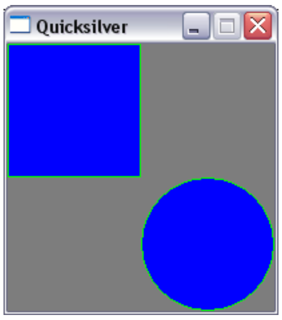


Figure 4.3: Image produced by listing 4.6

This State monad is not enough however, it needs to be combined with the IO monad to allow graphics to be drawn to the screen. Mixing monads in Haskell is quite challenging however the use of IO and State is common enough that Haskell provides a basic implementation. A lot of work is still required refining the provide StateIO monad to work with Quicksilver. A new parameterized type had to be created, listing 4.7, this combines State, Quicksilver (the name of the record discussed above) and IO, the a is the parameterized type which allows this new monad to return different data types for example returning the current colour.

All functions in the API output using this new type Quick and therefore must be used within the setup and painting function which are monads of this type. The monad and record are the bases of Quicksilver, supporting the graphics, handling inputs and managing the users variables. This common monad type is what allows Quicksilver to maintain purity, all actions are passing along the state, meaning they all have access and can change it but the user isn't forced to declare it each time, it is implicit when using the Quick monad.

Listing 4.7: Quicksilver monad type

```haskell
--StateT is a transformer monad to combine State with IO
type Quick a = State.StateT Quicksilver IO a
```

4.2.3 Program Inputs

As stated in section 3.2.2 Quicksilver is unable to use Processing function overriding method for attaching call backs. A number of alternate methods presented themselves. Use the IDE to parse the users text and insert default functions for unimplemented call-backs, present the user with all call-back functions, allow the user to explicitly attach functions to call back handlers, or manage the inputs and do not present the user with call-back functionality.

Presenting the user with all the possible functions where they often will not use any of them would be to overbearing on the user. Stepping in at compile time to insert code based on what the user wrote would look exactly like Processing however it would in fact be nothing like Haskell or Java and would present the user with a poor picture of how programming works. It would be showing a false image of how the language worked since it is a feature Haskell is actually unable to provide and is being faked behind the scene. Both of these were ruled out at the design stage.

Allowing the user to explicitly attach call-backs is a promising solution but makes the API more complicated. The intention is to keep the API as uniform and as simple as possible, this solution was perhaps the most feasible over not implementing call-backs at all. However for a number of reasons Quicksilver does not allow the user to implement call-backs. First it means the API is simpler and avoids explaining the notion of call-backs. Haskell does allow functions to be passed as values so implementation in that respect would not be to challenging.

However a key reason is that call-backs require the user to have a more complex model in their head of how the code will execute, issues such as concurrency and shared access would need to be handled. In fact implementing how the change of state in a call-back would affect code execution in the main graphics loop would be quite a challenge. Haskells demand of immutable variables means there would be no direct way to pass information between a call-back and the main loop if both were executing at the same time. The alternative would be to call the call-backs in order before the main drawing section, though this would not be a concurrent event handler and would be the same as if the programmer used an if block at the start of their code.

The most straight forward choice was to implement call-backs directly and store their affects in the state. The user can then access the state through standard API calls, for example getMousePosition simply returns the mouse position as it was last recorded in the state. This maybe a little out of date but for any reasonable frame rate would be unnoticeable. Key presses are handled by creating a list of keys pressed between frames, the user is then free to handle them as appropriate (perhaps only taking the most recent, or parsing all of them).

4.2.4 Managing User State

In Processing managing user state is quite straight forward. Variables the user declares globally become part of the class in which the user is writing, the values can then persist and be altered between frames as one would expect. Also changes the user makes to these variables obviously have immediate effect. Clearly Haskell is incapable of matching mutable variables in this manner however the user will need some way to create and store variables to allow animations to be generated.

The solution is to provide for the user a Record data type where they can put any variables they wish to be carried between frames. This record is then stored in the program state and passed around as described in the previous section. Listing 4.8 shows an example of a UserRecord, where the user has added extraVar and someName, they must also define defaultUserRecord which is where the start values of all the variables in the record are defined. screenX and screenY are required and allow the user to set the window size on start up.

Listing 4.8: Example of a UserRecord

```
data UserRecord = UserRecord{
        screenX :: Int,
        screenY :: Int,
        extraVar :: Int,
        someName :: String
}

defaultRecord = 200 200 42 "name"
```

To make it easier on the user their record is extracted from the overall state which the Quicksilver manages, and passed as a parameter to the setup and painting function, this spares them having to continually access the state explicitly to get any variable they wished to use. The one requirement then is that the user returns their updated version of their record. What the user will return is a new record with some of the same values as the old record, it is not the same variable it merely appears that way. This is important as access and updating the record during the users code must be handled in a specific manner. Listing 4.9 shows an example of accessing and updating the UserRecord.

Listing 4.9: Example of using UserRecord

```
painting dc record = do
        setFillColour (0, 0, 0)
        --Access x in the record
        rectPro dc ((x record),25) (50,50)
        --Add one to the old x and store in a new record
        return  record { x = (x record +1)}
```

The user might wish to update their record during the generation of graphics. This requires that the record be attached the same name as the old, or that a new name is used and used during all further access to the record. It is most likely the new record will be stored under the same name, the creation of a new record could also be achieved by some function created by the user. Listing 4.10 shows mulitple record updates.

Listing 4.10: Example of creating UserRecord's

```
painting dc record = do
        setFillColour (0, 0, 0)
        --access x
        rectPro dc ((x record),25) (50,50)
        --create a new UserRecord with the name record
        record <- record { x = (x record +10)}
        --access the value x in record
        rectPro dc ((x record),25) (50,50)
        --create a new UserRecord with the name newRecord
        newRecord <- record { x = (x record +10)}
        --access the value x in newRecord
        rectPro dc ((x newRecord),25) (50,50)
        {-return a record which is different
        from what we started with.-}
        return  newRecord
```

It is important to note here what exactly is taking place. While it might appear that a variable is being reassigned a value, that is not what is happening. In the monadic structure each line is nested within the line above it and all references map to the nearest line. In actuality the variable *record* is a new variable that is hiding the variable above it so that any further lines that access record see it and not the older version.

The syntax for this variable updating is not elegant and it would be nicer to somehow map the *record* <- *return* to something simple like *store*, however because the *record* on the left side of the <- it is a variable creation not a reassignment, it is not possible to remove or mask it. In truth it is against the grain of Haskell and is probably better left the way it is, it will encourage the user to better understand the use of variables in Haskell.

4.3 IDE

4.3.1 Program Compilation

Creating the IDE using wxHaskell was quite simple, except when compiling and executing the users code. The IDE is simply a wxHaskell text editor with an area for compiler output. In order to compile the users code a command line was launched and fed commands. This proved a challenge due to how the thread executing the command line is generated. processExecAsyncTimed is a wxHaskell function for executing asynchronous applications. Handlers are attached for processing the output and sending input to the running application. The aim is to launch a command line and allow it to send commands to itself and parses the resulting output.

processExecAsyncTimed returns a tuple containing information about the running application such as its PID (Process Identifier) and a stream for sending commands. The fact that these values are returned means it not possible to use them in the handlers attached to processExecAsyncTimed. Therefore wxHaskell's tool for storing mutable variables had to be used. Jumping through a series of hoops it is possible to create a reference to a variable in wxHaskell this is passed empty to the handlers and the information returned from processExecAsyncTimed is stored in the reference. This allows the handler to send commands to itself, parsing the output text looking for confirmation of compilation and then sending the command to run the executable. It obviously will have trouble if the reference is never set to a variable, this is possible since the executions are concurrent but in practise it does not happen.

The program code is generated by appending the users code to the Quicksilver library storing the result in a new file and compiling this file. This unfortunately leads to slow compile times, particularly the linking phase. The precise reason for the slow compile times is unknown and some further test could be done to ascertain where the problem is. Naturally compiling the library as one big file is not an ideal solution but due to the way in which the users code is interlinked into the library it is the least complicated approach and makes edits to the library much easier to implement. It would be possible through a series of module linking to pre-compile the library but this might require the code sections the user must provide to be altered. Time constraints have prevented further investigation of this possibility.

Section 5

Quicksilver Demonstrations

This chapter is an attempt to demonstate teaching functional programming through Quicksilver. It goes step by step through a number of examples and compares them to Processing. The full code all examples in both Quicksilver and Processing can be found in appendix B.

5.1 First Program

5.1.1 Moving Box

The graphics to be produced is a box moving across the screen which wraps around when it reaches the edge. The first thing to realise is we will need two boxes, when the box wraps we are drawing two boxes. So needing two boxes we need to store their two locations, we create an Int x and y in our record. Now we have to give them starting values, failing to do this will results in a compiler error.

Listing 5.1: Moving boxes: step 1

```haskell
data UserRecord = UserRecord{
    screenX :: Int,
    screenY :: Int,
    x :: Int,
    y :: Int
}

defaultUserRecord = UserRecord 200 200 (-50) 150
```

Now during setup we will set the background colour to white. Next we finally reach the painting function which is where the fun is, we set the colour we want to draw the boxes, black in this case, and then we draw two boxes. We use *x record* to extract the value of the variable named x in the Record named *record* and the same for y.

Listing 5.2: Moving boxes: step 2

```
setup record = do
                    setBackgroundColour(255, 255, 255)
                    return record

painting dc record = do
                    setFillColour (0, 0, 0)
                    rectPro dc ((x record),25) (50,50)
                    rectPro dc ((y record),25) (50,50)
                    return (update record)
```

Now the important bit, we need to return our variables but we want them to be replaced with the position we will draw the boxes in the next frame. To do this we need a function for each variable. This function will take our record and will update the appropriate variable, either incrementing or resetting the value. Notice that we can chain these together as each returns a new record which feeds into the next, eventually being the value which painting returns.

Listing 5.3: Moving boxes: step 3

```
update record = (moveX (moveY record))

moveX record = if(x record == 350)
                    then record {x = -50}
                    else record {x = (x record )+1}

moveY record = if(y record == 350)
                    then record {y = -50}
                    else record {y = (y record )+1}
```

The code will look quite similar in Processing (listing B.2 in appendix B) as this is quite a simple program. This demo highlights the important differences about how variables are managed.

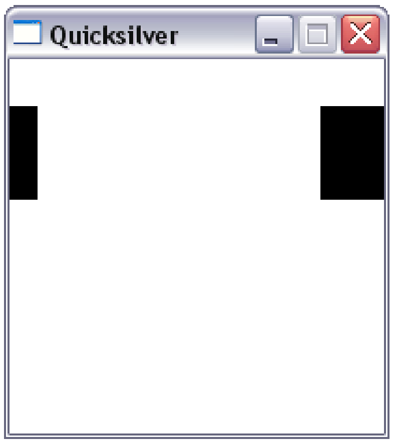


Figure 5.1: Image produced by Moving Boxes demo

5.2 Interaction and Lists

5.2.1 Colourful Circles

The aim of this demo is to produce a chain of circles which size and colour depend on the speed of the mouse. This will introduce lists and inputs during the program. So first we will need a few variables, a tuple to store the x and y of the previous mouse position and a list to store the information to draw the circles: a tuple which contains: a tuple for the position and an Int for the speed. Along with these we need to provide starting values.

Listing 5.4: Colourful Circles: step 1

```
data UserRecord = UserRecord{
          screenX  :: Int,
          screenY  :: Int,
          circles  :: [((Int,Int),Int)],
          pMouse  :: (Int,Int)
}

defaultUserRecord = UserRecord 640 360  [] (320,180)
```

In the setup we store the last position of the mouse ready for the first loop. Then we get to drawing the circles, to do this we need to reverse our list and then run down it, drawing a circle for each entry. First we need to write how we are going to draw each circle though. drawConCircle takes the xy position and the speed at which to draw the circle, using the speed it varies the colour of the line and the fill and then it draws the circle with that radius.

Listing 5.5: Colourful Circles: step 2

```
setup record = do
                  --Do Setup Here
                  xy <- getMousePosition
                  return record{pMouse = xy}

drawConCircles2 dc (xy, spe) = do
                  setLineColour (spe*2,spe*2,spe*2)
                  setFillColour (spe*2,spe*3, spe*4)
                  circlePro dc xy spe

speed (ox, oy) (nx, ny) = (abs ( nx - ox)) + (abs (ny -oy))
```

The circle drawing function is then mapped down our list using Haskell's predefined map, the result is a list of StateIO actions which can be executed by calling sequence_ another Haskell predefined function. This draws our list of circles.

Listing 5.6: Colourful Circles: step 3

```
painting dc record = do
        --Expanded over multiple lines to fit page
        revCir <- return  (reverse (circles record))
        sequence_ (map (drawConCircles2 dc) revCir)

        xy <- getMousePosition
        record <- return record{circles =
                         (xy, (speed (pMouse record) xy))
                                        :(circles record)}
        return record {pMouse = xy}
```

Finally we need to add the next circle information, first we get the mouse position, then we calculate the next circle entry. This is done by storing the position and computed speed, and attaching it to the head of the existing list, we also need to store what is now the old mouse position. This list can grow infinitely and ideally we should add some guard against circles with speed 0 and when the list is too long dropping old value, however it serves its purpose.

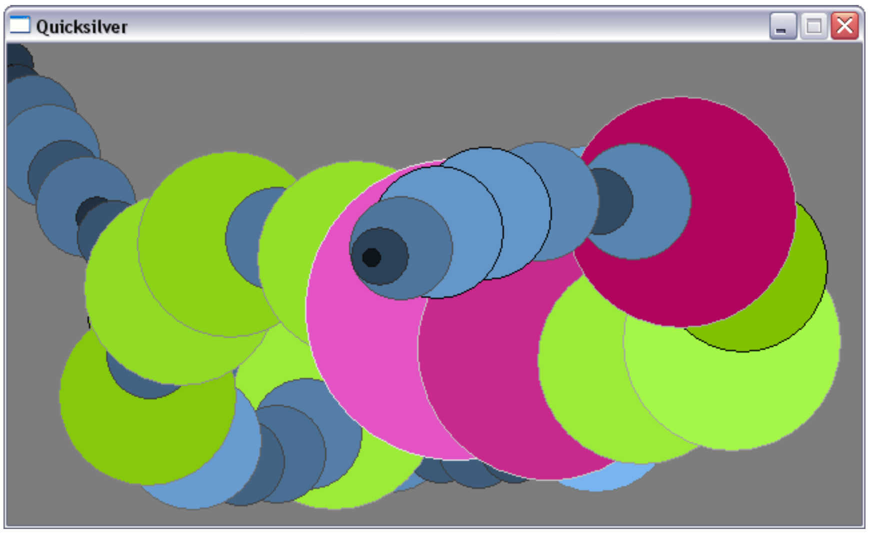


Figure 5.2: Image produced by Colourful Circles demo

This example serves well at introducing lists, how the user accesses inputs and some of the useful inbuilt Haskell functions. Map and Sequence_ are quite easy to code but the user will find themselves using them in every program so Haskell provides them for ease of use. Processing manages this in fewer lines by not flushing the screen after each loop (listing B.4 in appendix B), it therefore does not need to store information about where each circle is. Otherwise Processing would need a series of arrays storing information about where circles should be drawn which would easily grow larger then the Haskell version.

5.3 Infinite Lists and Functional Programming

5.3.1 Sine Waves

The purpose of this demo is to produce a moving sine wave and so learn about infinite lists and some functional programming fundamentals. Again as usual the first thing we need are some variables. In this case we will use a list of lists of Int, that is each element in the list will be a list of integers. This will be our wave. Now we use our setup function to produce the sine wave. We are going to use random numbers to determine the amplitude and frequency, Quicksilver provides a function which produces a random number between a given range. Note that randomNumRange is a StateIO action and therefore must be used on its own line.

Listing 5.7: Sine Wave: step 1

```haskell
data UserRecord = UserRecord{
          screenX :: Int,
          screenY :: Int,
          wave ::[[Int]]
}

defaultUserRecord = UserRecord 400 200 []

setup record = do
          a <- randomNumRange (10, 60)
          b <- randomNumRange (100,300)
          return record {wave = (createWaveS a (8*6.32/b) 0.02)}
```

Having generated the amplitude and frequency, we can now create the wave, this is where it gets more complicated. First we create an infinite list, the function createWaveS will never terminate, each time it is called it will call createWave and then call itself again with a slightly increased off-set, it attaches the result of createWave to the result that its recursive call returns. createWave is the function that generates the sine wave, it too is an infinite list. It uses a counter to determine a value for the sine wave, it then attaches value to the results of a recursive call itself, each recursive call increases the counter moving the sine wave along. To try and make this clearer think of it this way, we have an infinite list representing each pixel across the width of the screen, and each pixel has an infinite list associated with it, which maps its position through infinite time. Each frame we move down this list in time, moving the pixel, and we map across the list to produce a series of points across the window.

Listing 5.8: Sine Wave: step 2

```
createWaveS :: Double -> Double -> Double ->  [[Int]]
createWaveS amp dx dx2 = (createWave amp dx2)
                                    : (createWaveS amp dx (dx2+dx))

createWave :: Double -> Double -> [Int]
createWave amp dx =(truncate ( (sin dx )*amp) )
                                    : (createWave  amp (dx + 0.02))
```

Having generated the wave we now move to the painting function. Here we want to do two things, first display the instance of the wave in time, and then we want to remove this instance and move down the list to the next one. In this way we create the animation of the wave moving across the screen. Quicksilver has a useful function called polylinePro which draws a line connecting all the points in a list in the order they appear, it cannot be fed an infinite list unfortunately as it will immediately attempt to draw the lines and so get stuck in an infinite loop of execution. polylinePro takes a list of tuples which are the x and y coordinate of each point, we currently have the y position so we need to generate the x position. This is where we can use a number of list operators provided in Haskell to make our job easier.

Listing 5.9: Sine Wave: step 3

```
painting dc record = do
        polylinePro dc (
                zip xLocations (map (100+)(map head (wave record)))
                                )
        return record {wave = (map (tail) (wave record))}

xLocations = [0,8..400]
```

The first thing we wish to do is get our instance in time, to do that we map head down our infinite list, this returns the head of all the lists into a new list. This is our frame, but all the values will be around zero we need to move it to the centre of the screen. To do that we map 100+ down the list, this increases all the values by 100. Now we have our y positions where we want them but we are still missing the x. Haskell provides a nice function called zip which takes two lists and zips them into a list of tuples, pairing each value. So we need a list of x values which are the points on the screen where we want to draw our lines. Haskell again comes to rescue providing the .. operator which extrapolates the values that should go in a list based on the first 2 the coder provides. The important thing to note is that although this could be infinite that would produce an infinite list of tuples which we cannot pass to polylinePro, therefore this is a handy way to limit our list to the visible screen. Now we have our x and y list we simply zip them together and draw the result. Our next step is the update the list for the next frame, to do this we simply map tail down our infinite list, this function drops the first item in each list and returns the new list, in this way we move through time.

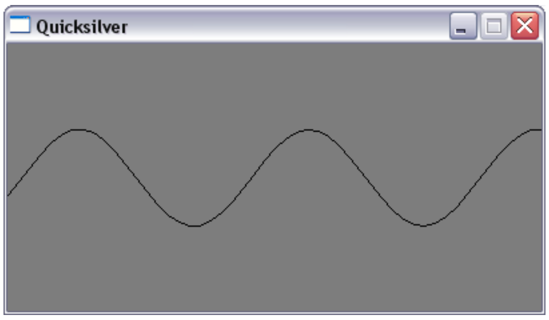


Figure 5.3: Image produced by Sine Wave demo

With this demo we have learned about infinite lists and the power they possess and we have seen a number of tools Haskell provides to make it easy to work with them. We have also learned how counter intuitive Haskell can be at first glance. In order to create an animation we had to compute all of its locations before we used them, it seems a little weird. Then one would conclude we should zip the x locations at the start to spare the computation later on. However the truth is that our infinite list only produces one value each frame, all of the computation actually takes place where you think it would, when the line is drawn. If we had moved zip to the setup all the computation would still take place in the same location. Our infinite list is really a function to produce the next value which can be accessed each frame for infinity.

Now when we compare this code to Processing (listing B.6 in appendix B) we see an interesting result, it is nearly 50% smaller. All of the messing around with for loops and creating variables is not needed in Quicksilver, the coder only needs to worry about the bits of the code that actually produce results. We achieve this by using laziness to assign the calculation directly to the list. This avoids having to initialize the list in each loop, we simply move down the list and the result is waiting for us. This is a big departure from Processing, this demo combines drawing an arbitrator long list of points and taking advantage of laziness to defer calculations, neither of these are provided for in Processing.

5.4 Analysis

5.4.1 Quicksilver as a Learning Tool

The aim of Quicksilver is to act as a learning tool, to introduce people to Haskell. The three preceding demos show a nice progression, first introducing the user to the fundamentals of creating variables and drawing, second working with lists, finally infinite lists. Haskell can be a challenging language to work with, mainly because of its counter intuitive nature. Often when introducing Haskell most texts start with standard functions, they particularly avoid Monads since they are significantly different from the rest.

Programs like ghci and hugs provide interpreters that can produce output without using the IO Monad. Quicksilver starts with Monadic actions, they allow the most feedback (IO to produce graphics) and they are the easiest to work with since they are both sequential and almost mutable. It is very hard to do anything complex purely within monadic action so the user is eventually forced to try the more complex functional programming.

The three main challenges in learning Haskell are, handling purity, being aware of laziness and working with monads. The first one the user meets is purity, it is significantly different from imperative style. This is the part Quicksilver tries least to hide, anything more complicated then a single variable or a single function will require chaining functions and variables. The user will immediately be forced to leave the monad framework and program in a clearly pure way.

Laziness is less a road block and more a tool for writing elegant programs, without realizing its power the user would be forced to construct barriers to avoid infinite recursion. The third example clearly shows that infinite recursion is a solution and not a problem, it needs to be understood to be used properly. Monads are by far the hardest thing to manage in Haskell, for that reason Quicksilver doesn't require the user to create any new monadic action or use anything other then Quicksilver's Quick monad. By focusing only on the Quicksilver monad which combines the power of State and IO the user is given the easiest introduction to their use in Haskell, while clearly showing the two sides of Haskell, monadic and non-mondaic.

Section 6

Conclusions

6.1 Critical Analysis of Work Completed

6.1.1 Comparison to Processing

The last 3 demos from the previous sections are based on Processing examples. It is clear that Quicksilver can produce similar work, though its quality is slightly less than Processing, particularly curves. Quicksilver does not implement as wide a range of graphical functionality as Processing provides. Time constraints are clearly a factor here, with limited time it was not possible to match Processing's graphics quality or quantity, instead Quicksilvers use as a learning tool was a priority. The previous examples are a clear outline how Quicksilver can act as a learning tool and in that way it certainly matches Processing.

Quicksilver also makes it easier to visualise how functional programming works, the power of constructs' such as infinite lists and the toolkit needed to access them. It can be hard at first to understand how functional programming works since it differs so greatly but by providing examples of imperative and functional producing the same result the key areas of interest are highlighted.

One area which Quicksilver does not touch on is monads, beyond the provided StateIO monad no attempt is made to introduce monads or how they are used. This is due to their more complex nature and insufficient time to develop a good example using them. However by avoiding monads Quicksilver is better able to encourage a functional style of programming, since monads can give a false impression of supporting imperative programming. Much like how Processing has gone on to support other features such a video and network, the use of monads could be explored to provide that kind of functionality.

Processing allows for everything java provides including creating new classes, this is quite handy and is used in many of their examples. Quicksilver does not support adding addition modules within the IDE and so its not possible to gain the same affect. A lot of these differences however represent the fact that Processing aims to be a professional language while Quicksilver is still very much a prototype. With work many of these gaps could be bridged and this will be discussed in Section 6.2.

6.1.2 Graphics Library

When beginning the project there was insufficient time to fully check each of the available graphics libraries. wxHaskell and Gtk2Hs were the most prominent, wxHaskell looked a good choice, its examples revolved around animation, and it seemed to provide everything needed. By the time the project was completed two issues began to appear, first loading images was not fail-safe and second the graphics produced were rough.

Loading images caused trouble if the file name was incorrect, wxHaskell handles the exception internally by displaying a dialog box. This prevented the program loop of Quicksilver detecting the issue, which leads to an infinite loop of error messages without the opportunity to close the window. This would cause annoying problems for someone new to programming and is not an ideal solution. This may be caused either by the way the event loop is executed or specifically by how wxHaskell handles the error, at this stage it is not apparent if changing library would solve the problem.

The second issue is the resolution of the graphics produced, they are at times quite poor, particularly curves. They produced excessive sharp angles and are particularly obvious in animation, where curves can be quite jaggy and jumpy. The solution to this may be to use OpenGL to produce and render the graphics within wxHaskell but at this stage it would require considerably more work.

Neither of these problems cause Quicksilver serious trouble but any further development should investigate gtkHs to see if it too would suffer the problems that wxHaskell is presenting.

6.1.3 API Layout

One key aspect that was a known issue from the start was how to separate the users code from the library. Since the library needed access to the users code and the user needs access to the library, neither could survive without the other.

Haskell makes this more of a challenge since variables are immutable and so the user has to provide a default for their variables before the library can compile.

There were a number of possible ways to handle this issue but by the time most of the library was written it was too late. This resulted in the current system of attaching the user's code directly to the library code and compiling them together. This is not ideal as it has slow compile times and allows the user to access parts of the library which should not really be exposed. If I were to go back and start again I would spend more time designing a better layout for how the library would support the users code and would present itself to the user. A redesign might allow multiple modules to be linked, much like classes in Processing. There are a number of possible solutions using Haskell's support for module linking and type parameterization but none could be elegantly integrated at the later stages of development.

6.2 Further Work

One of the big areas where further improvement could be achieved is in the IDE, particularly the compiler output. In its current state the compilers output is sent straight to the text area and since the code is part of a bigger file the line numbers are wrong. It would be beneficial to the user if the line number were appropriate to their code. Also errors in the users code can lead to errors in the library, these errors are presented to the user which can be misleading. The error messages themselves can be quite obtuse, it would be worth determining the most common errors and handling them in a more forgiving manner. Processing does something similar, common errors such as missing ';' and ')' are highlighted independent of the compilers output.

The API could also be expanded to include OpenGL commands, this would greatly expand the range of graphics that can be produced. Also as mentioned previously better support for blitting images is needed. The hope would be that by supporting the graphics using OpenGL the produced curves and shapes might be smoother. Along with smoother graphics, smoother animation could be achieved by monitoring the frame rate. By implementing a time delay between frame calls based on previous times, swings in execution time would be presented as a uniform frame rate. In the current situation frames are refreshed faster and slower depending on execution time of the particular frame.

As you may have noticed all of the API calls which involve shapes require *DC a* as a parameter. This is the drawing context which wxHaskell uses to figure out how and where to draw shapes on the screen. It is parameterized over a type to allow it to act as a layer of abstraction over low level parameters that the library requires. It would be ideal to store DC within the program state, this way the user would not be required to add it to every function, the implementation could take care of it behind the scenes. A lot of time was spent trying to find a way which this reference could be abstracted away. By the time this project was completed no way was found. There is no technical reason why this could not be hidden from the user but a much more low-level knowledge of wxHaskell and how it the DC value is created is required to find a solution.

Some additional work could be done by testing Quicksilver with users both experienced and not experienced with Haskell to get feedback about how easy it is to use. Overall the project as outlined is completed but like any such project continued work will see it improve, to that end Quicksilver is in the process of being uploaded to Haskell.org along with a supporting web page. Hopefully the Haskell community will find Quicksilver a useful tool.

6.3 Summation

The title of this project is a Processing like domain specific language in Haskell, from this I drew two key points, make graphics easy for the user and focus on teaching the user Haskell. In order to do both it was necessary to provide an IDE as well as a domain specific language.

Is a tool such as this useful? Much of the time spent at the start of this project was spent learning Haskell. Haskell has quite a steep learning curve which had to be overcome. It is therefore clear that a tool which makes learning easier and more enjoyable would have been an very useful at the time.

Quicksilver achieves this by using Processing as an example of such a tool. The result is a powerful language, the graphics are slightly slower to render then Processing but as this is a prototype that is less of an issue. Shapes can be produced in a similar way to Processing, which is simple and has served well as the bases for Quicksilver.

Animation is the key difference, because Quicksilver is functional animating an object requires a different approach. It is not easy to have a bunch of variables that are updated each loop like in Processing. The more elegant way involves using lists and this is clearly shown in chapter 5.

Quicksilver uses Haskell and wxHaskell as a firm base to build a domain specific language on. The advantage of this is that it frees the design from the ground work of syntax and graphics detail. Haskell provides sufficient abstraction that wxHaskell could be removed and replaced with a different library, the existing code would need to be re-worked but user generated code would not need to be significantly changed. The DC a issue mentioned previously would be the only change and as stated would be better off removed anyway. This is the advantage of embedding a domain language within a large language, the new language has more room to evolve and grow by using the base language to support it.

The same can be said for basing Quicksilver on Processing. It would have been possible to develop a graphics API from scratch, particularly focusing on functional style graphics. By basing Quicksilver on Processing we gain a firm example of easy to use graphics and we can ensure a familiarity for those with experience of Processing or OpenGL. This has the advantage of giving the user something familiar to work with but slowly taking it way as they try to do more complex things. The functional style will eventually show up.

The key aim of not hiding Haskell is important, at times it can seem to make things more complex for the user but because of how different it is, hiding too much would not present the user with enough of the picture to learn. By combining code writing and execution into an IDE like Processing, Quciksilver also makes it much easier to get started and to see the results. The IDE was not part of the original project but it soon became apparent that without it, it would be much harder to present Quicksilver in an easy to use manner, not everyone is familiar with command line compiling.

In conclusion Quicksilver has achieved its goals, as a prototype it compares favorably with Processing and with further work could match it. A number of example have been created with it that show how easy it is to us and it encourage a functional style and with that a different way to think about programs.

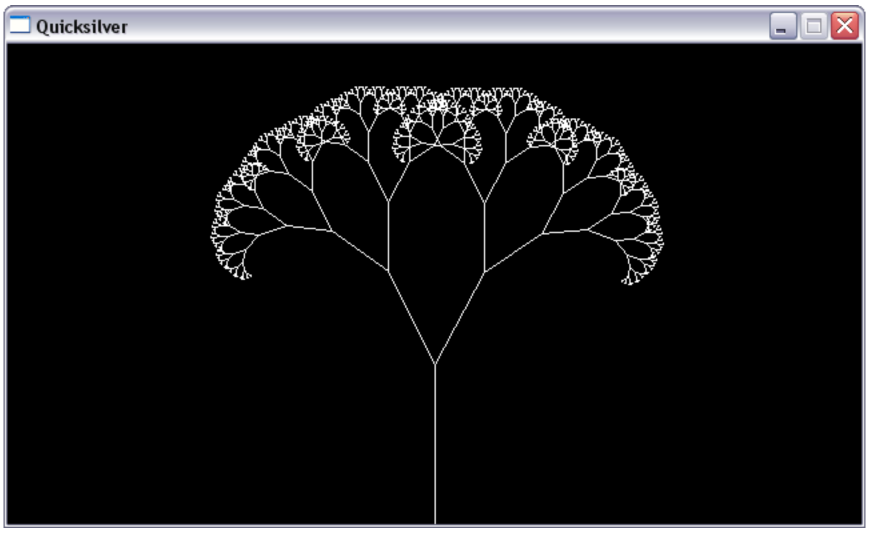


Figure 6.1: Fractal Pattern produced by Quicksilver, listing B.7.

Bibliography

[1] John Goerzen Byran O'Sullivan and Don Stewart. *Real World Haskell*. O'Reilly, 2008.

[2] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[3] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[4] Daan Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, September 2004.

[5] MIT Media Laboratory. Design By Numbers. `http://dbn.media.mit.edu/`, April 2010.

[6] Simon L. Peyton Jones Paul Hudak, John Hughes and Philip Wadler. A History of Haskell: Being Lazy With Class. In *The Third ACM SIGPLAN History of Programming Languages Conference*. HOPL-III, June 2007.

[7] Processing.org. Why Java? Or why such a Java-esque programming language? `http://www.processing.org/faq.html`, April 2010.

[8] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[9] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

Appendix A

Quicksilver API

Graphic types:

```
setFillColour :: (Int, Int, Int) -> Quick ()
getFillColour :: Quick (Int, Int, Int)
setLineColour :: (Int, Int, Int) -> Quick ()
getLineColour :: Quick (Int, Int, Int)
setFill :: Bool -> Quick ()
getFill :: Quick Bool
setBackgroundColour :: (Int, Int, Int) -> Quick ()
getBackgroundColour :: Quick (Int, Int, Int)
setLinethicknes :: Int -> Quick ()
setCentreMode :: Bool -> Quick ()
```

Graphic Shapes:

```
linePro :: DC a -> (Int, Int) -> (Int, Int) -> Quick ()
ellipsePro :: DC a -> (Int, Int) -> (Int, Int) -> Quick ()
circlePro :: DC a -> (Int, Int) -> Int -> Quick ()
arcPro :: DC a ->(Int, Int) -> (Int, Int) -> (Double, Double) -> Quick ()
PointPro :: DC a -> (Int, Int) -> Quick ()
polygonPro :: DC a -> [(Int, Int)] -> Quick ()
polylinePro :: DC a -> [(Int, Int)] -> Quick ()
rectPro :: DC a -> (Int, Int) -> (Int, Int) -> Quick ()
drawTextPro :: DC a -> String -> (Int, Int) -> Double -> Quick ()
drawPixelMap :: DC a -> [(Int,Int,Int)]->(Int,Int)->(Int,Int)-> Quick ()
```

Input:

getRightClick :: Quick Bool

getLeftClick :: Quick Bool

getMousePosition :: Quick (Int, Int)

getKeysPressed ::Quick [Char]

Utility:

stringToTerminal :: String -> Quick ()

printToTerminal :: (Show a) => a -> Quick ()

randomNumRange :: (Num a) => (a, a) -> Quick a

blitImage :: DC a -> String -> (Int, Int) -> Quick ()

Appendix B

Demo code listings

Listing B.1: Moving boxes Quicksilver

```
data UserRecord = UserRecord{
    screenX :: Int,
    screenY :: Int,
    x :: Int,
    y :: Int
}
defaultUserRecord = UserRecord 200 200 (-50) 150

setup record = do
                        setBackgroundColour(255, 255, 255)
                        return record

painting dc record = do
                        setFillColour (0, 0, 0)
                        rectPro dc ((x record),25) (50,50)
                        rectPro dc ((y record),25) (50,50)
                        return  (moveX (moveY record))

moveX record = if(x record == 350)
                        then record {x = -50}
                        else record {x = (x record )+1}

moveY record = if(y record == 350)
                        then record {y = -50}
                        else record {y = (y record )+1}
```

Listing B.2: Moving boxes Processing

```
int x = -50;
int y = 150;

void setup()
{
        size(200,200);
}

void draw()
{
        background(255,255,255);
        fill(0,0,0);
        rect(x,25,50,50);
        rect(y,25,50,50);
        x++;
        y++;
        if(x > 350)
                x = -50;
        if(y > 350)
                y = -50;
}
```

Listing B.3: Colourful circles Quicksilver

```
data UserRecord = UserRecord{
        screenX :: Int,
        screenY :: Int,
        circles :: [((Int,Int),Int)],
        pMouse :: (Int,Int)
}

defaultUserRecord = UserRecord 640 360  [] (320,180)

setup record = do
                --Do Setup Here
                xy <- getMousePosition
                return record{pMouse = xy}

drawConCircles2 dc (xy, spe) = do
                setLineColour (spe*2,spe*2,spe*2)
                setFillColour (spe*2,spe*3, spe*4)
                circlePro dc xy spe

speed (ox, oy) (nx, ny) = (abs ( nx - ox)) + (abs (ny -oy))

painting dc record = do
        --Expanded over multiple lines to fit page
        revCir <- return  (reverse (circles record))
        sequence_ (map (drawConCircles2 dc) revCir)

        xy <- getMousePosition
        record <- return record{circles =
                        (xy, (speed (pMouse record) xy))
                                        :(circles record)}
        return record {pMouse = xy}
```

Listing B.4: Colourful circles Processing

```
void setup()
{
        size(640, 360);
        background(102);
        smooth();
}

void draw()
{
        variableEllipse(mouseX, mouseY, pmouseX, pmouseY);
}

void variableEllipse(int x, int y, int px, int py)
{
        float speed = abs(x-px) + abs(y-py);
        stroke(speed);
        ellipse(x, y, speed, speed);
}
```

Listing B.5: Sine Wave Quicksilver

```
data UserRecord = UserRecord{
        screenX :: Int,
        screenY :: Int,
        wave ::[[Int]]
}

defaultUserRecord = UserRecord 400 200 []

setup record = do
        a <- randomNumRange (10, 60)
        b <- randomNumRange (100,300)
        return record {wave = (createWaveS a (8*6.32/b) 0.02)}

createWaveS :: Double -> Double -> Double ->  [[Int]]
createWaveS amp dx dx2 = (createWave amp dx2)
                                 : (createWaveS amp dx (dx2+dx))

createWave :: Double -> Double -> [Int]
createWave amp dx =(truncate ( (sin dx )*amp) )
                                 : (createWave  amp (dx + 0.02))

painting dc record = do
        --Generate Graphics Here
        polylinePro dc (
                zip xLocations (map (100+)(map head (wave record)))
                                 )
        return record {wave = (map (tail) (wave record))}

xLocations = [0,8..400]
```

Listing B.6: Sine Wave Processing

```
int xspacing = 8;
int w;

float theta = 0.0;
float amplitude;
float dx;
float[] yvalues;

void setup() {

        size(400, 200);
        frameRate(30);
        colorMode(RGB, 255, 255, 255, 100);
        smooth();
        w = width + 16;
        amplitude = random(10,30);
        float period = random(100,300);
        dx = (TWO_PI / period) * xspacing;
        yvalues = new float[w/xspacing];
}

void draw() {
        background(0);
        calcWave();
        renderWave();
}

void calcWave() {
        theta += 0.02;
        for (int i = 0; i < yvalues.length; i++) {
                yvalues[i] = 0;
        }
        float x = theta;
        for (int i = 0; i < yvalues.length; i++) {
                yvalues[i] += sin(x)*amplitude;
                x+=dx;
        }
}
```

```
void renderWave() {
        noStroke();
        fill(255,50);
        ellipseMode(CENTER);
        for (int x = 0; x < yvalues.length; x++) {
                ellipse(x*xspacing, height/2+yvalues[x], 16, 16);
        }
}
```

Listing B.7: Fractal pattern Haskell

```
data UserRecord = UserRecord{
        screenX :: Int,
        screenY :: Int,
        theta :: Double
}

defaultUserRecord = UserRecord 640 360 0

setup record = do
                --Do Setup Here
                setBackgroundColour (0,0,0)
                return record

painting dc record = do
                --Generate Graphics Here
                a <- getMousePosition
                record <- return record { theta = computeAngle (fst a)}
                setLineColour (255,255,255)
                linePro dc (320, 360) (320,240)
                branch dc 120 (320,240) (3.1615/2.0) (theta record)
                return record


computeAngle :: Int -> Double
computeAngle x = radians ((90.0*(fromIntegral x))/640.0)

radians :: Double -> Double
radians x = (x*3.1615)/180.0
```

```
branch dc pos (x,y) th dth= do
  if(pos < 1.0)
    then return ()
      else do
          linePro dc (x,y) (endPoint (x,y) (pos*0.66) (th+dth))
          linePro dc (x,y) (endPoint (x,y) (pos*0.66) (th-dth))
          --Should be one line:
          branch dc (pos*0.66)
                      (endPoint (x,y) (pos*0.66) (th+dth))
                      (th+dth) dth
          --Should be one line:
          branch dc (pos*0.66)
                      (endPoint (x,y) (pos*0.66) (th-dth))
                      (th-dth) (-dth)
          return ()

endPoint :: (Int, Int) -> Double -> Double -> (Int,Int)
endPoint (x,y) len th =
            (x + truncate ((cos th)*(-len)) ,
                 y+ truncate ((sin th)*(-len)))
```

Appendix C

CD

The accompanying CD is laid out in the following folders and files:

GHC Installer.exe
A Windows installer for the Glasgow Haskell Compiler.

wxHaskell folder
wxHaskell binaries for Windows. Includes a bin file which registers the library.

IDE.exe
The executable for Quicksilver, requires access to the neighboring bin folder.

bin folder
Contains the IDE source and the library source. The library source does not compile, it is intentionally missing the functions the user must provide. The library file is used by the IDE to build the users program. Both files require wxHaskell.

Quicksilver.pdf
A copy of this report.