ABSTRACT

This paper presents a generalized system for graphical IO that views the graphical environment as a separate IO device, for Haskell and other functional languages. Haskell uses monads in its IO system but these are considered difficult and stimulate an imperative way of thinking. This paper shows why an implementation of the so-called stream model provides a solution to these problems. We demonstrate the feasibility of the generalized system and how it can be used to achieve platform-independent graphical IO, by presenting a prototype system that uses the generalized system, and the graphical capabilities of a browser as the graphical IO device. By viewing the browser as a more complex IO device, it is also possible to use its keyboard and mouse information-capturing features and communicate them to the program along the same IO connection as the graphical IO.

Keywords

Haskell, graphical IO, IO, functional programming, functional programming language, platform-independent graphical IO, complex IO device, browser, browser as IO device, generalized IO system.

1. INTRODUCTION

Graphical Input/Output (IO) is one of the key elements within any computer program. It is used to show the graphical state of the program to help the user interact with the program. Programmers can use IO within their programs using an IO model that is specified within the used programming language. Functional programming languages are used to define functions and parameters as opposed to imperative languages where the computer receives a list of instructions. This results in IO having to be handled from a different perspective in functional programming than in imperative programming. This paper demonstrates that functional graphical IO can be handled in a generalized system. Haskell is a popular functional programming language and is considered the target language.

One of the key features of functional programming is referential transparency, which is the lack of side-effects within a program. Usually, the IO within imperative programs is full of side-effects which are functions that change their semantics based on the state of the program. While this creates a contradiction, successful approaches have been taken to tackle this problem such as monads [21] in Haskell [3], Uniqueness Typing [6] in Clean [16] and the Eventloop in Amanda [1] and an older version of Haskell [9].

The Haskell committee discussed IO [8] at the hand of three IO models [10]. These three models are: the stream, system, and continuation model. These were later discussed again in a work presenting an addition to the typing system [17]. Monads, uniqueness typing, the event-loop, and others each use one of the IO models to perform IO. Rather than discussing each approach, we will discuss IO using the models.

Haskell is the target language which uses monads as its IO system. An implementation using monads, which uses the system model, seems to be easiest as it is possible to use all of the tools Haskell has for monads. However, monads seem more difficult to understand than most other solutions [17]. They also tend to stimulate an imperative way of thinking [4] [12] and the IO monad is action oriented while processing IO is actually data oriented. Researching a possible alternative to present the graphical IO using the three IO models as a basis is needed.

This leads to the following question:

• How would a generalized system of graphical IO look like in Haskell using either the stream, system or continuation model?

This question can be decomposed into the following sub-questions:

• What are the characteristics of graphical IO?
• What are the characteristics of the stream, system, and continuation IO model?
• Which of the stream, system, and continuation IO model should be used to present graphical IO in a functional language?
• What are the requirements for a generalized system of graphical IO using either the stream, system, or continuation model?

To show the usage of the generalized system, a prototype system has been added to this work. This prototype system shows that it is possible to achieve platform-independent graphical IO by using a browser as the graphical IO device. Browsers can be found on most operating systems and use a standardized set of languages defined by the World Wide Web Consortium [23] from which...
they occasionally stray. Due to this, any implementation of a graphical IO device using a browser, is platform-independent. The goal of the system would be the ability to express programs that would translate incoming IO messages to instructions for the target IO device in the form of an ongoing IO message. The prototype system will be validated using the requirements which are set by the generalized system.

This paper is structured as follows: First the analyses of graphical IO and the stream, system and continuation model are given. Using these analyses, the arguments for the choice of the stream model are presented. With the choice for the model made, the requirements for the generalized system are presented. This is followed by the implementation and validation of the generalized system. Next similar solutions and the prototype system are discussed. The discussion of the prototype system is started by an analysis on graphical IO and browsers followed by the implementation of the system and ended with a small discussion of its validity. The paper is concluded with a discussion of the results and a look on future work.

2. ANALYSIS OF GRAPHICAL IO

Every programming language has some form of input/output (IO) processing to communicate with the external world. In imperative languages, primitives are given in order to receive and send data to devices e.g. a keyboard, mouse, graphics card, hard drive and memory DIMM. The ANSI C programming language uses functions e.g. fopen, fclose and fgetc to respectively open and close a file and retrieve a character from an open file [11]. Typically IO exhibits the following characteristics:

- Streams - In most programming languages (C, C++, Java and Haskell) streams are implemented to handle long lists of incoming messages/elements so they can be processed element per element. An example of this is the contents of a file in C, C++ and Java. Haskell has an InputStream module to simulate similar behavior.

- Order - An IO device specifies messages in a certain order. This can cause problems for synchronous IO solutions that could result in deadlocks. An example of a deadlock scenario is explained in the masterthesis by R.J. Rorije [17]. When an outgoing message is dependent on multiple incoming messages in one device while one or more outgoing messages are dependent on an answer in the other device a data race will occur as both devices are waiting for an incoming message and the system will deadlock.

- Error Handling - Numerous situations can cause faulty communication with an IO device. An IO solution typically presents these situations by using error codes e.g. C or exceptions e.g. Java & Haskell.

Graphical IO normally takes place within a single computer system between the graphics-card and the processor or even the screen and the processor and is done in a similar fashion. The goal of graphical IO is to communicate graphical state changes. These can consist of a stream of pixel changes directly to the screen or communications to the graphics-card. However, low-level libraries e.g. DirectX [14] and OpenGL [19] define relatively high-level functions for the graphics-card such as vertices that define an area which is filled automatically. These high-level functions can be used to express a set of primitive shapes e.g. rectangle, circle and text which can be used to express more complex shapes e.g. a message box. The primitive shapes can be drawn using functions which we shall refer to as hooks from now on.

The graphics-card's function is to output a picture. As such, no external data is captured by the graphics-card which has to be send to the processor. The processor never has to wait for external data from the graphics-card. Even so, it might be necessary to send certain event information such as error codes. This is why the order characteristic has to be taken into account. A solution for this potential problem is to ensure that for each message from the graphics-card, another message is not needed to continue the program. So if an error code is sent to the processor, the processor might make a note that such a message has been sent and send a reply, but it must not be dependent on further information from the graphics-card to evaluate the rest of the program.

This results to the following characteristics of graphical IO:

- Primitives or Streams - Graphical IO must state a certain set of shape primitives which can be used to create more complex shapes or must have the possibility to handle streams of pixel changes.

- Hooks - Graphical IO must state a certain set of callable functions(hooks) that need to be available to draw the primitives and, if possible, the composite shapes.

- Order - Each message must be treated as a singular dialog. In other words, each message from the IO device might be the last on the subject so dependencies on further information is not allowed.

- Error Handling - Graphical IO must state the possibility of perceiving errors and being able to act upon them.

Normally IO also handles the problem of multiple sources and destinations. As the graphics-card simplifies any multiple destinations to a single destination and the graphics-card itself can be considered a singular source, this characteristic of IO is non-existent in graphical IO.

3. ANALYSIS OF THE IO MODELS

In order to create a system for graphical IO a model must be used to present the IO to the programmer. The three models [10] [17] that will be discussed and chosen from are the stream, system, and continuation model.

3.1 Stream model

The stream model is based on using a channel of IO messages, represented as a list. Each of these messages is processed and translated into an answer. These answers are appended to a response-channel, which is also represented as a list. This results in mapping elements from an ‘incoming’ list to a response and appending them to an ‘outgoing’ list. A state can be added throughout the program to use information from earlier messages in future messages. IO channels can be terminated by using a finite list as representation for the incoming IO channel which ends upon channel termination. Depending on the implementation of the model, the incoming and outgoing lists could also be represented as infinite lists. In this case, a specialized message may be used to show that the
IO channel is closed. An example to illustrate the stream model using a finite list approach with state:

```haskell
data State = State {hasSetup :: Bool}
```

beginState = State False

start :: [Message] -> [Answer]

start i = process beginState i

process :: State -> [Message] -> [Answer]

process s [] = []

process s (i:is) = answer ++ rest

where

(\(\text{answer}, s'\)) = handleOne s i

rest = process s' is

handleOne :: State -> Message ->

   ([Answer], State)

handleOne s m | hasSetup s = (read m, s)

| otherwise = setup s m

read :: Message -> [Answer]

read "Hello!" = ["Right back at you!""]

read "Bye!" = ["Nice meeting you!"]

read m = ["I did not expect that"]

setup :: State -> Message -> ([Answer], State)

In this example a state has been added and the list is lazy. The start function adds a beginstate to the invocation of process. Process uses handleOne to determine the correct answer for each incoming message. HandleOne returns an altered state which is used in the recursive invocation of process. Due to the lazy nature of both lists, functions that use start can use already processed answers while the IO channel list has not been processed completely. Also, messages are only processed when available and the list (and further evaluation) blocks until that moment.

Characteristics

The following remarks can be made about the Stream model:

- Simple - Using this model is simple and straightforward. It is very compact and uses only one 'setup' function as overhead.

- Non-general implementation - The general concept of this model does not specify any general approach to creating channels. Also, the lazy lists that represent the IO channel are unique and different from other lists. While the type is just a list, existing implementations use foreign functions written in other languages to push received messages unto the list and send processed answers. Also, there has to be some concept of converting incoming IO data to Haskell datastructures. If a standard is created, it will be hard to modify/extend it [12].

- Data - This model does not specify any 'actions' such as readMessage or sendMessage. Any incoming messages are considered data that needs to be transformed.

- Blocking - A paper [12] suggest that this model can create a datarace relatively easily by using multiple messages to create an answer. However, the same holds true for any model that uses a blocking IO primitive which is most IO where first data has to be fetched before it can be used. File and network are a couple of the IO contexts that are implemented using a blocking IO primitive.

This model has been implemented in an older version of Haskell [9]. It has been substituted by a system using a System model approach and the monads programming structure [8] so it may seem odd to revisit this model. However, we shall prove that these issues can be resolved in section '3.4 Decision for the Stream model'.

3.2 System model

The system model describes an overall state of the entire system. This state is modified by functions that require information from the state to accommodate any side-effects. This results in a state being passed through each IO function containing all the system changes. However, this models turns out to be very inefficient in multi-threaded solutions [10], something that may be added in future work. Nonetheless it is possible to use this model by adopting a small variation to the model called Uniqueness Typing [6]. The resulting model is nearly the same. The difference can best be explained using an example:

```haskell
readTwoLines :: File -> ([Char], File)
readTwoLines file0 = ([11, 12], file2)
where

(11, file1) = readLine file0
(12, file2) = readLine file1
```

In C a file is represented by a special datastructure that keeps a pointer to where in the file the program has read. When another character has been read, the pointer is moved the size of that character. When looking at this from a functional perspective, one can note that the file datastructure has changed. This changed file datastructure can be seen as a 'new' file datastructure even though both datastructures use the same physical file. By keeping the reference to each version of the file datastructure unique, referential transparency is kept. As such, the function readTwoLines takes a file and returns a new file that can be used in further IO. ReadTwoLines uses the file and reads from it twice, each time receiving a 'new' file. The same approach can be used for any form of IO. The system model with Uniqueness typing will henceforth be referenced to as the system model. An example to illustrate the use of this model with a message based IO system:

```haskell
type Message = [Char]
```

```haskell
start :: Channel -> Channel

start c = case a of

  "ExitOk" -> c''

  _ -> start c''

where

(c', m) = readMessage c

a = handleM m

c'' = sendAnswer c' a

readMessage :: Channel ->

   (Channel, Message)

readMessage c = case char of

  \n -> (c', [])

  _ -> (c'', char:m)

where

(c', char) = read channel s

(c'', m) = readMessage c'
```

```haskell
read :: Channel -> (Channel, Char)
```
handleM :: Message -> Answer
handleM m | m == "Hello" = "Bye"
| m == "Setup" = "Ok"
| m == "Exit" = "ExitOk"
| otherwise = "error"

sendAnswer :: Channel -> Answer -> Channel

In this example start receives a channel from which to read from. A message consists of the text until a newline character. The start function reads one message using readMessage, translates it to an answer using handleM and then sends the answer back over the channel using sendAnswer. If the answer was to exit, then start returns the channel. Otherwise it will recursively invoke start itself, to read the next message. Each read or send action takes a channel and returns a new version of it. Each version is also only referenced once, as is per the Uniqueness Typing system.

Characteristics
The following remarks can be made about the System model:

- Multiple versions - Multiple versions can live within one function. Therefor, great care has to be taken to preserve the uniqueness condition of each reference. While other languages e.g. Clean [16] use specialized language constructs to preserve the uniqueness, Haskell does not.
- Efficiency - Each mutation on the state causes a different copy of that state. This creates a lot of data redundancy. It would be more efficient if handling each version would result in a reference to the former state accompanied with the changes. However, to create such an implementation the language would have to be able to maintain such references. As Haskell cannot do this natively, a workaround using the current IO system with references has to be used.
- Program Flow - The program flow is clear. Functions in the program can only be resolved when the state is passed by former functions. These functions only pass the state after execution.

3.3 Continuation model
The continuation model describes IO implementations that define a set path throughout the program. Each function receives a function that should be invoked upon completion of the function with the calculated result. When combining functions this way, the result is a(n) (enormous) composite function. When the composite function is invoked, the result is calculated before the function k can be evaluated as a whole. The reason why this model has been used in the past is that the program flow is set; even for lazy functional programs. In this example first readMessage is evaluated, then handleMessage and only then sendMessage. Only after sendMessage has been evaluated will the function k be evaluated. This holds also true for the implementation of the functions readMessage, sendMessage and handleMessage.

Characteristics
The following remarks can be made about the continuation model:

- Evaluation order set - The global flow of the program is set and clear.
- Error handling and recovery - Any errors can be handled locally within each function. Example: When a message cannot be sent, it is saved in the state. The next time `sendMessage` is invoked, the buffered messages in the state will be send as well.
- Types are complicated - The addition of an extra function that is called upon end of the function results in a more complicated type.
- Code bloat - There is a significant amount of extra code needed in both the type declarations and function declarations to follow the continuation style. The resulting code is harder to read and comprehend.

3.4 Decision for the Stream model
The three IO models are analyzed in the previous section’s but the decision to use one has not been made. The core of the decision is how the IO will be presented to the programmer. It is known that the IO consists of messages from an IO device that need to be processed and translated to answers that are sent to that IO device. The answers are needed after each message. Each of the three IO models can be used as a solution. However, a solution using Haskell’s current IO system might be easier to implement as Haskell already has the tools to simplify development. The current IO system, as mentioned before, is a variation of the System model(without Uniqueness Typing) using monads [12] [2].

An implementation using monads would consist of using the IO monad. This monad passes a pseudo state to enforce evaluation order. This pseudo state represents the entire system and any changes to the system, such as

```
handleMessage :: State -> [Char] ->
    (State, [Response]) -> r
handleMessage' :: State -> [Char] ->
    (State, [Response])
sendMessage :: State -> [Response] ->
    (State -> r) -> r
sendMessage s resps = \k ->
    k $ sendMessage' s resps
sendMessage' :: State -> [Response] ->
    State
```

```
readMessage :: State -> ((State, [Char]) -> r) -> r
readMessage s = \k ->
    k $ readMessage' s
readMessage' :: State -> (State, [Char])
```
reading and writing. This state, however, does not literally contain the system state but is rather a token that is changed upon each change to the system to enforce evaluation order.

Using monads can be difficult and stimulates an imperative way of thinking [12] [17]. IO consists of translating data -in other words, IO is data-centered. The IO monad enforces the creation of an action to process each data-message, an action-based approach, instead of functions that translate each data-message, a data-centered approach. The IO monad stimulates an imperative way of thinking due to the action-based nature of the IO monad. A functional program is a list of definitions, not actions. A look at the other models explained might lead to a more data-centered and functional stimulating way of thinking approach.

An implementation using the continuation model would force the programmer to create bloated code and complicated function types so this model is not preferred.

An implementation using the stream model would be simple. The incoming list of messages could be elegantly processed one by one while adding answers to the outgoing list. The data-centered approach of the model ties in perfectly to the issue at hand. There is, however, the problem of a non-general implementation. Specialized lists would have to be added to the language and the data-structures that are using in these lists cannot be easily extended or modified. This last point can be solved by giving the translation of data to datastructures as a responsibility to the programmer. Each message would consist of a list of characters (a string) and the programmer would have to write functions to translate that to, for instance, a keyboard press data-structure or a mouse-click data-structure. This would also allow for a common "input stream list" to be implemented within the language that can be used with any IO device and so also solve the specialized list problem.

By also adding the possibility of passing a state through functions, it is possible to save ‘notes’ for further evaluation. This makes sure that the order characteristic of graphical IO is not violated.

An implementation using the system model would result in the need for a state such as a channel and functions to read and write to and from the channel. This channel would allow to save ‘notes’ for further evaluation just as the stream model would. This ensures that the order characteristic is also not violated in this model. There is, however, the point of efficiency. While this is a non-argument for small programs that would only have a few copies of the current state at a time, this might become a serious problem for bigger programs with large states. Both the system and stream model would result in a good solution but the stream model is possible to do so without the efficiency problem of the system model.

While all three models would result in a correct solution, the stream model’s data-centered approach and simpleness without any efficiency problems is the favored way to go. This is why we have chosen for the stream model to handle the IO.

4. REQUIREMENTS OF GENERALIZED SYSTEM

Using the analysis of graphical IO and the choice for the stream model, the following list of requirements can be constructed, which any system using a graphical IO device will have to fill.

The System will...

1. ...provide a Stream representation of IO
   (a) ...provide a (infinite) list for incoming messages from an IO device
   (b) ...provide a (infinite) list for outgoing messages from an IO device
   (c) ...provide means for automatically pushing incoming messages from the target IO device to the incoming messages list
   (d) ...provide a means for automatically sending outgoing messages to the target IO device from the outgoing messages list
   (e) ...provide a means for connecting to an IO device which results in an incoming messages list and an outgoing messages list
   (f) ...use a string representation for any incoming or outgoing IO message
   (g) ...present a method to retain information from messages for future evaluation of the program

2. ...provide a graphical IO device to output a graphical state

   The graphical IO device...
   (a) ...will be able to receive messages using a string representation
   (b) ...will be able to send messages using a string representation
   (c) ...will be able to change the graphical state of a monitor
   (d) ...will provide graphical primitives or the ability to process pixel changes
   (e) ...will provide hooks
   (f) ...will be able to handle errors
      i. ...will be able to send error code messages
      ii. ...will be able to perceive errors
      iii. ...will be able to recover from errors either on its own or using the instructions from a received message

5. IMPLEMENTATION OF GENERALIZED SYSTEM

The generalized system actually has two parts: the functional program side, which tells the graphical IO device how to act, and the graphical IO device. A generalized implementation of the graphical IO device cannot be given as that is dependent on the graphical IO device itself and the environment in which it is run. A generalized implementation of the functional program side, however, is partly possible. It is possible to give a part of the IO system which will interact with a programmer-written program, together forming the functional program. As stated before, Haskell is the target language which uses the system model and monads to form its IO system.

During the project we have not been able to find a way using the IO action system of Haskell to produce lazy monadless lists which can be read in one thread and modified in another. This has to do with the fact that the IO monad is strict and Haskell does not allow the programmer to introduce thunks to make something lazy. A thunk is a special Haskell data type that is a place holder for some value or computation. It is the essence of laziness in Haskell.
When a thunk is evaluated the computation or value will be calculated or retrieved. However, if it is not yet available the program will block at that part of the function until it is possible and try to resolve (if any) dependencies for that computation or value.

In order to construct the system, we had to use a workaround. As can be seen in the Stream model code example, a function handleOne was used. The function to take one incoming message and translate it into the answer(s) is one that can be expected in any implementation of the stream model. Using Haskell’s IO System it is possible to use such a function within the IO monad. This can be illustrated with the actual implementation below:

```haskell
type IOMessage = [Char]

start :: IDevice ->
a ->
(a -> IOMessage ->
([IOMessage], a)) ->
IO

start dev begSt func =
  handle func begSt conn
  where
      conn = connect dev

handle :: (a -> IOMessage ->
([IOMessage], a)) ->
a -> Connection ->
IO

handle func state conn =
do incoming <- readMessage conn
  let
      (ans, state') = func state incoming
      sndact = map (sendMessage conn) ans
  sequence sndact
  handle func state' conn

connect :: IDevice -> Connection

readMessage :: Connection -> IO IOMessage

sendMessage :: Connection -> IOMessage ->
IO
```

The implementation defines a function to connect to an IO device (connect) to read a message from the connection (readMessage) and to send a message onto the connection (sendMessage). It also defines a function to read, handle and send a message (handle) and a function to start the connection and the program (start). It needs a programmer-written function that uses a state (a) and a message (IOMessage) to produce a tuple of a list of answers (IOMessage) and a new state (a).

As the connect, read and send functions are IO device dependent, it is not possible to give a generalized solution. Even though it is not possible to create the traditional stream IO system with two lists, it is possible to use the function that translates a single message to a list of answers as has been done with the implementation.

### 6. VALIDATION OF GENERALIZED SYSTEM

In this section, we prove that the given implementation fulfills the set requirements of the generalized system. The validation is given in table 1 using the requirements from the ‘4 Requirements of Generalized System’ section.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a)</td>
<td>This was not possible. However a workaround using a function that translates a single message to a list of answers was used</td>
</tr>
<tr>
<td>1b)</td>
<td>This was not possible. However a workaround using a function that translates a single message to a list of answers was used</td>
</tr>
<tr>
<td>1c)</td>
<td>As can be seen in the implementation, each incoming message is translated using the given programmer-written function func and the messages are read using the sendMessage function</td>
</tr>
<tr>
<td>1d)</td>
<td>Each of the answers provided by the programmer-written function func is sent using the sendMessage and sequence functions</td>
</tr>
<tr>
<td>1e)</td>
<td>As the connection to each IO device is different, this is something the programmer must give. However, the system can use the function and the needed type is given</td>
</tr>
<tr>
<td>1f)</td>
<td>The system uses the IOMessage type for any messages. This is a synonym for the list of characters type which is the string type</td>
</tr>
<tr>
<td>1g)</td>
<td>The system allows for a parameter to passed through each evaluation of the read, handle and send cycle. This ensures that programmers are able to pass a store of information to future evaluations of the cycle.</td>
</tr>
</tbody>
</table>

Table 1 shows that each requirement is met in the implementation. However, as this is a tool for a programmer to create a program and some functions are IO device dependent, not all definitions for functions can be given. Due to this, it is not even possible to give a definition of the IO device to communicate with, but the requirements that the IO device has to fulfill are given. In a later part of the paper, we shall show that this generalized system is usable to communicate with an IO device to do graphical IO.

### 7. SIMILAR SOLUTIONS

Doing graphical IO is not something new in the functional programming world. Effort has been made to create libraries that can be used in a functional programming language to do graphical IO to achieve some goal. For example, game and GUI libraries exist and are being used:

- Functional Reactive Programming(FRP) - A long existing issue in the functional world was the fact that the time dimension is non-existent. A programmer would have no control on when certain actions would take place or how long ago a certain event has taken place. Especially for animations and other time-dependent graphical structures, it is important to have some form of time dimension. Functional Reactive Programming adds the time dimension to a functional language and an example of this used for graphical purposes is the language Elm [5]. It gives an asynchronous approach to handle graphical IO combined with a time dimension. The main difference with this work is the way how the IO is
perceived. In this work, the graphical part of the computer system is viewed as another external IO device while with Elm the graphics device is part of the program. As is presented in section ‘8 Example of a Prototype System’, the connection from the program to the graphical IO Device, can actually connect to a more complex IO device. This complex IO device can handle and send information on multiple tasks. An example of this would be another computer that can send keyboard strokes, mouse positions, internet data and even timer ticks. Elm can also handle any of these tasks but views them as part of the system and as such they are tangled within the program as opposed to the modular approach of the generalized system we present. Elm’s approach forces the creation of code where multiple purposes are handled within the same part of the program, while the modular approach enables simplicity by reserving parts of the program to a single purpose. This will be presented in the section ‘8 Example of a Prototype System’ where the keyboard functionality, generalized system and program logic are all kept in small separate files.

- **wxHaskell** - wxHaskell [13] is a GUI library written for Haskell that is build on top of a multi-platform C++ GUI library called wxWidgets. In both wxHaskell and our system it is possible to create interactive GUIs but, as with FRP, wxHaskell views the graphical device as part of program and this results in the similar issue of code complexity.

8. EXAMPLE OF A PROTOTYPE SYSTEM

In this section, we present a prototype, which can do platform-independent graphical IO using the generalized system. We use a browser as a complex IO device that can communicate to the generalized system using websockets. The result is communication with an IO device that can capture keyboard strokes, capture mouse positions and show graphical output using only one connection. The resulting system can be used to write a graphical program that translates incoming IO messages into outgoing IO messages. The system itself will use such a program to communicate with a graphical IO device. First an analysis on graphical IO and browsers is presented to show that it is possible to use a browser for graphical IO.

8.1 Graphical IO and Browsers

Browsers use a set of languages to describe a webpage. A webpage consists of structured code and browsers can interpret the code to display the webpage. Languages include HTML5 (for structure), Javascript (for client-side dynamic programming) and CSS (mark-up of the structure) with each having its own functionality and they are defined by the World Wide Web Consortium [22]. HTML5 also defines the Canvas object [22] which is used to show a graphical state. Javascript is used to invoke the Canvas API to draw on the canvas. The Canvas API uses both pixel manipulation primitives as shape primitives. Below a summation is given of all the graphical IO characteristics and how using a browser can fulfill these:

- **Primitives** - The standard Canvas primitives can be drawn using Javascript.
- **Streams** - It is possible to design a stream-based approach using the pixel and ‘ImageData’ (large collection of pixels) primitive. It is, however, not initially supported as a stream.
- **Hooks** - The Canvas API consists of hooks which can be invoked to draw primitives. Also, there exist frameworks that provide simpler default API access and add extra functionality e.g. jCanvas [7] & KineticJS [18].
- **Order** - A Javascript script can be written to decode IO messages and call the respective hooks. Care must be taken when building this script to ensure the order characteristic and that each incoming message must not rely on another incoming message before an action can be taken.
- **Error Handling** - The protocol between the client and server can be extended to include error messages. Error messages are captured client-side and processed server-side.
- **Platform-Independence** - As browsers exist on most operating systems and the browser is implemented using a set of standardized languages e.g. HTML, CSS and Javascript, any implementation of the graphical IO device in a browser can be considered platform-independent.

Using these arguments, a platform-independent graphical IO device using a canvas object can be designed. However, we have not yet discussed how this ‘device’ can actually communicate with the outside world. Javascript has native access to a websocket API that is also specified by the World Wide Web Consortium. Websockets is a bidirectional communication line between the browser and a server and by using websockets as a connection method, both the browser and Haskell program can be connected. The analyses so far have led to the design of a system that can do graphical IO with a browser. In figure 1 a simple overview is given of this system.

![Figure 1: Simple overview of prototype system](image)

8.2 Implementation of Prototype System

The prototype system consists of two parts. A client side which houses the browser, javascript functions and canvas object on the one hand and the server side, which houses the Haskell generalized system, websocket implementation and programmer-written IO program on the other hand. JSON [15] is the used protocol between the client and server to encode messages. To simplify things at the canvas API, a canvas framework called jCanvas [7] is used. This results in the detailed overview of figure 2.
8.2.1 Server Side

The Haskell side of the system uses the generalized system and the websocket implementation to handle any IO messages. The following code snippet is an addition to the generalized system:

```haskell
type IOMessage = JSONMessage
start :: (a -> IOMessage -> (IO IOMessage, a)) -> a -> IO ()
start eh beginState =
  WS.runServer ip address port $ application eh beginState
application :: (a -> IOMessage -> (IO IOMessage, a)) -> a ->
  WS.PendingConnection -> IO ()
readMessage :: WS. Connection -> IO IOMessage
readMessage conn =
  do msg <- WS.receiveData conn
     putStrLn (T.unpack msg)
     let string = T.unpack msg
     request =
       stringToJsonObject string
     return request
sendMessage :: WS. Connection -> IOMessage -> IO ()
stringToJsonObject :: [Char] -> JSONMessage

We have used the websocket library built by J. van der Jeugt [20]. This websocket implementation also comes with a server, which is used to connect to the browser. This can be seen in the function `start`. The function `application` handles the connection and calls the generalized function `handle` to start the program. We have used a JSON implementation as intermediate step to parse the incoming IO messages. These are parsed from the textual messages and then given to the programmer. For this system we have included the mouse and keyboard incoming messages. An example of how the keyboard is handled is shown in the following code snippet:

```haskell
class FromJSON a where
  fromJsonMessage :: JSONMessage -> a

data Keyboard = KeyPress KeyboardButton

instance FromJSON Keyboard where
  fromJsonMessage (JSONObject ms) = KeyPress button
  where
    JSONString button = retrieveError buttonS ms

As can be seen, an incoming IO message in JSON format can be parsed to a keyboard datastructure which can then be used in the programmer-written IO program. Each of the 'devices' that are connected to the program can have a `FromJSON` instance to parse the incoming JSON message to the preferred datastructure. All these device datastructures are combined into the following code snippet:

```haskell
data JSONMessage = JSONObject [JSONMember]
data JSONMember = JSONMember
  [Char] JSONMessage

data Request = ReqKeyboard Keyboard

instance FromJSON Request where
  fromJsonMessage (JSONObject ms) |
    keyboardS ==
      type
        ' =
          ReqKeyboard (fromJsonMessage obj)
  where
    JSONString type' =
      retrieveError typeS ms

retrieveError :: [Char] ->
  [JSONMessage] ->
  JSONMessage

A type variable in the JSON message tells us to which device the message belongs to and is parsed accordingly. The resulting `Request` datastructure can then be used in the programmer-written IO program as the structure that represents incoming IO messages.

The outgoing messages consist of graphical state changes using the canvas graphical primitives. An example of how these are handled is shown in the following code snippet:

```haskell
class JSONAble a where
toJsonMessage :: a -> JSONMessage

data Response = ResGraphical Graphical
data Graphical = Draw GObject Groupname
data GObject = GObject
  { name :: Name ,
    prim :: Primitive ,
    children :: [GObject] }
data Primitive = Circle ...

instance JSONAble Response where
  toJsonMessage (ResGraphical graph) =
    JSONMessage ...
```
The data structure that represents outgoing IO messages is called Response. These can consist of all the ‘devices’ that ‘listen’ to the program for instructions. In this example a graphical command called Draw is implemented, which draws the accompanied graphical data structure. An instance from the class JSONAble defines how a Response object can be defined in JSON terms, in this case a JSONObject. The GObject is a data structure that defines the different graphical objects. These objects can house the different graphical primitives, which are defined in the Primitive data structure.

A possible implementation for the programmer-written IO program using these examples is:

```haskell
processIOMes :: State ->
  IOMessage ->
  (([IOMessage], State)
processIOMes s m = (map toJsonMessage 
  resps , s')
  where
  (resps , s') = process s 
  (fromJsonMessage m)
processRequest :: State -> Request ->
  ([Response], State)
processRequest s 
  (ReqKeyboard (KeyPress 'c'))
  = 
  ([ResGraphical (Draw Circle "")], s)
```

This example draws a circle when the ‘c’ button on the keyboard is pressed.

8.2.2 Client Side

The client side uses a Javascript websocket implementation to communicate with the server, a canvas object with the jCanvas framework to show graphical output and an interface we developed to connect the two and capture mouse and keyboard activity.

The implementation is very straightforward and will not be discussed within this paper.

8.2.3 Validity of Prototype System

While this prototype system works, it has yet to be discussed if it abides by the requirements set forth by the generalized system. The validity of this prototype is tested using the requirements of the generalized system in section 4. In this case, the graphical IO device is the browser environment.

The stream representation of IO is provided. Instead of providing lists for incoming and outgoing messages, the function handle is implemented that takes a function eh, which translates a single incoming message to a list of outgoing messages and takes a state of type a, which it translates to a modified state of type a. This is discussed in section 5 Implementation of Generalized System. The function application is used to read messages using the function readMessage, map each incoming message over the function eh and send the result using the function sendMessage. The function start automatically connects to the browser. Each message is communicated as a string using a websocket connection.

The graphical IO device is also provided. It is able to send and receive messages using a websocket connection which is implemented in Javascript. Using the canvas object it is possible to output and modify the graphical state. The canvas object also provides graphical primitives which are available at the server side and the respective hooks are provided using the jCanvas framework. Even though it is not provided in the code snippets, error codes can be sent as regular messages with the error type. The server side is able to send instructions to the browser as an error code is perceived. The browser is able to recover from errors using these instructions or it is possible to recover on its own. In the event of a non-recoverable problem, the connection is terminated.

These arguments prove that the prototype system fills every requirement set by the generalized system.

9. CONCLUSION

We have demonstrated that it is possible to create a generalized system of graphical IO using the stream IO model with Haskell and what characteristics and requirements it must fulfill. The generalized system has been validated using the requirements which were drawn based on the analyses of graphical IO and the stream model. We have shown that by using this generalized system in combination with browsers, it is possible to achieve platform-independent graphical IO by viewing the browser as a separate graphical IO ‘device’. It is also possible to view the browser as a more complex IO device, which results in the ability to also capture keyboard strokes and mouse positions. The prototype system was validated using a more informal argumentation.

10. FUTURE WORK

The generalized and prototype system show that it is possible to do graphical IO. It might be possible to include other forms of IO as well, by using a general way of multiplexing multiple IO devices into one connection. A method of this has been used with the browser which is a complex IO device but it might also be possible to multiplex multiple IO devices into one connection or find a way for a multi-threaded system in which each separate IO device is handled by one single-threaded program.

The solution might also be fruitful for educational endeavors. By using the stream IO model, a very simple solution is reached. The simplicity is also acknowledged by R.J. Rorije [17]. As said before, monads often offer a challenge for beginners. When first learning a functional programming language, using a stepping stone such as this system might help orient a beginner in the functional world. He can create IO programs using the simple concept of streams and after gaining some experience he can dive into the challenging world of monads.

11. ACKNOWLEDGEMENTS

The author would like to thank Robert E. de Groote and Jan Kuper for supervising the entire project. They have provided invaluable counsel from when this project was just an idea until the finalization of the paper. The author would also like to thank Philip Hölzenspies for his helpful remarks on monads and abstractness and Ivan Perez Dominguez for his helpful remarks on Functional Reactive Programming.

12. REFERENCES


