Next to Godliness

A tutorial on the Concurrent Clean IO libraries, with several examples

Glenn D. Strong
Glenn.Strong@cs.tcd.ie

April 9, 1998

Abstract

Concurrent Clean is a pure lazy functional programming language, with a syntax reminiscent of the Haskell family of languages. There is a comprehensive library for performing IO operations such as building interfaces. Unfortunatly, there is little in the way of introductory documentation for this library. This report attempts to rectify this by presenting several programs in the literate programming style, and discussing their implementation.

1 Introduction

Normal IO (that is, IO in typical imperative languages) features the concept of state. For instance, when reading bytes from a file in a language like C there is an implicit state maintained by the system, such as the concept of a *current file position*, which can be modified by the *fseek* function. In a functional environment the concept of implicit state is anathema. In a language like Concurrent Clean there is no way to specify a variable which is implicitly modified when it is used. Traditionally this approach makes it very difficult to write IO intensive programs in a functional style.

1.1 The Clean approach

The approach taken to IO in Clean is both simple and elegant. It is a variant on the explicit state passing mechanism, Each function which will make changes to the state of the "world" takes an additional parameter representing the state of the appropriate part of the "world", and returns another object of the same type which represents the new state of the IO system. For instance, a function to read a line from a file might have a signature

$$readLine\ ::\ \mathbf{FILE} \rightarrow (\mathbf{FILE}, \mathbf{string})$$

However, once such a function has been used, the IO system which is returned is the only valid one, and the IO system state which was passed in is no longer valid. Consider a graph:

$$(newState, line) \equiv readLine\ oldState$$

Once this graph has been evaluated, the **FILE** object *oldState* is no longer valid. The current file position, for instance, has been advanced by the number of characters read. Ideally, there should be no way to access the state variable *oldState* once it has been used, because *newState* is the only valid description of this file.

Clean provides a mechanism to ensure that certain objects will only be used in a "single threaded" manner by declaring them to be *unique*. An object of unique type may only be used once in any expression. This ensures that only the correct value will be used, as only the most recently calculated will be available. With this in mind, the correct type signature for the *readLine* function is

$$readLine :: \star\mathbf{FILE} \rightarrow (\star\mathbf{FILE}, \mathbf{String})$$

The annotation "$\star$" indicates that the particular argument is a unique object. In any function which uses *readLine*, the use of the $\star$**FILE** parameter is the *only* allowable use of that object; it is a type error to use it a second time. If another **FILE** object is required, then the one returned by *readLine* must be used. This will not only sequence the IO operation (by forcing *readLine* to be fully evaluated before the new **FILE** object can be returned), but it will also ensure that the correct file object is always used.

This style of passing unique parameters is central to the way IO is performed in Clean. There are a number of special objects which must be passed into special library functions to perform IO. The **FILE** type is one of them, and refers to a file which has been opened using another system call. We shall see in the coming sections how to create and use these objects.

1.2 Curried functions for IO

Readers experienced with functional programming may skip this section, which recaps some basic principles of functional programming that are heavily used in the remainder of the tutorial.

Let's say we want to draw a line. The type of a function to do such a thing might be:

$$draw\ ::\ Window\ Line\ \star IO \rightarrow \star IO$$

(This is not the real type of the function in Clean, by the way). So, if

$$draw\ w\ l\ io$$

is an application of the function to a window w, a line l and a suitable IO object io, then its type is $\star IO$. What is the type of this?

$$draw\ w\ l$$

The answer is that it is a function which takes a single unique IO object, and transforms it to a unique IO object! This is called a curried function. We convert a function to another function of fewer arguments by supplying some of the arguments but not others.

Why is this useful? Consider a function *apply* which takes a list of function of type $\star IO \rightarrow \star$ and an IO object, and which works similarly to the standard $fold$ functions. Now, if we wanted to draw three lines in a window, we could make up a list of curried IO functions:

$$[draw\ w\ line_1, draw\ w\ line_2, draw\ w\ line_3]$$

and use it with *apply*. These curried functions can be very convenient when we are writing IO programs. You will frequently find IO functions with the IO components left out, producing new functions to be applied later.

1.3 The Console and the GUI

Although all IO in Clean works the same way, there are two distinct modes of operation for Clean programs. The "console" mode is the normal operating mode for a Clean program. In this mode a program does not open or read from any files, but it can be useful for testing parts of a program. All of the programs in this tutorial will use the other mode of a clean program, called "World" mode. In this mode, the *Start* function of a program is of type

$$Start :: \star\mathbf{World} \rightarrow \star\mathbf{World}$$

As far as the IO types are concerned, **World** is the mother of all objects. A program written in world mode will receive one object of type $\star$**World** which will contain all the required information about the state of the world outside the program.

With this in mind, let us examine out first example program, which will reveal some of the basic usage of these concepts.

2 Convivial greetings

First we will introduce a Clean version of a classic computer science program, updated by the addition of a graphical interface. The program will create a single window, and will display a message in the window welcoming the user to the world of functional programming.

Figure 1: Main window for the Hello application.

2.1 Clean IO Model

The clean IO model makes extensive use of unique types to preserve the referential transparency of the programming language. This approach produces a distinctive code style which can appear, at first glance, quite complex. In reality, most programs follow quite a simple model.

The entire clean IO system is conceptualized as an object of type **World**. The world contains everything that is not internal to the program, such as display devices, the filesystem, etc. IO operations work by making changes to this World object, which will cause changes to be made to the *external* world (e.g. causing a window to appear on the screen). Referential transparency is preserved by the use of *unique types*.

A typical program will usually begin by extracting the GUI *events* from the world. The *OpenEvents* function produces two objects: the GUI *events*, which contains all of the users potential interation with the program (mouse clicks, keyboard actions, etc.), and the remainder of the world (which contains the filesystems, etc). In a standard GUI based program, the

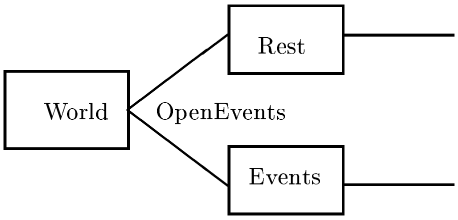

Figure 2: Initial partitioning of the world.

remaining processing focuses on these events. If the program performs file IO, this is handled separately by operations on the remainder of the world. The first step in a GUI program is to specify the *initial* interface. This is the user interface as it appears before any events are

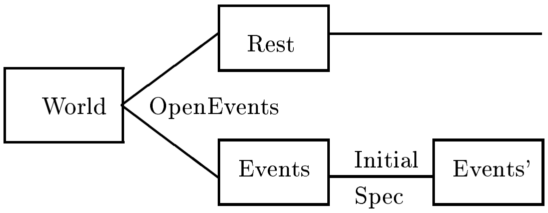

Figure 3: specification of initial GUI.

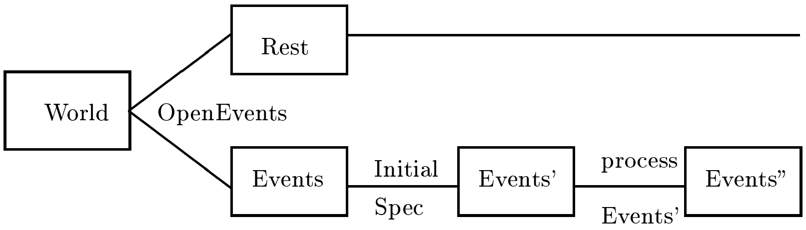

Figure 4: Processing the users interaction.

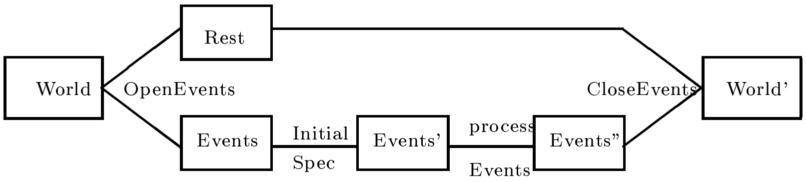

Figure 5: The reunification of the World object.

processed. It is normal here to specify things like menus, and perhaps any windows which the program opens as it is started.

The program will then begin processing the users interactions. This is conceptually similar to the main loop in, say, an X toolkit program, in that all of the processing is handled by the IO library function *StartIO*, which plays a similar role to the X Toolkit function `XtAppMainLoop`. This function operates over all of the users interactions, and acts as a kind of "central dispatcher", calling the users *callback* routines whenever the appropriate interaction has taken place. For instance, if a menu option "Open file" is selected by the user, then the hypothetical callback *LoadNewFile* might be called. This callback might display a standard file requester, and load the file selected in this. At some point, the user will indicate the interaction is complete, usually be selecting a "Quit" option from a menu, or by pressing the window systems "Go Away" button (a button on the window decoration which causes the window to close). The special IO function *QuitIo*, contained in the IO library, is used at this point to terminate the processing of the users events. The program can now tidy up and shut down. Part of this "tidying up" involves reuniting the parts of the **World** object which were split up at the start of the run. The entire program is of type *Start* :: ⋆**World** → ⋆**World**, so we must produce a complete world object once more before the program can reasonable shut down. The *CloseEvents* function can unite the events which remain after the users interaction terminated with the remainder of the world.

2.2 Wrapping

As well as the usual standard environment, we shall need to import several modules containing definitions for the Clean IO system. The exact modules required will, obviously, vary depending on which operations you wish to perform.

5a ⟨*library imports* 5a⟩≡ (9)

import *StdEnv*, *deltaEventIO*, *deltaIOSystem*, *deltaWindow*
import *deltaSystem*, *deltaPicture*, *deltaFont*

2.3 Creating windows

Consider for a moment a function which will create a new window to be shown on the screen. How can we write such a function? Well, Clean provides a standard way to do such things in the IO library. If we have a function of type

5b ⟨*HelloWindow type* 5b⟩≡ (9)

HelloWindow :: *State IO* → (*State*,*IO*)

Where *State* is some type we have defined for objects which contain program state, and *IO* is a standard type for the programs IO state, then we can adjust the programs state to achieve the effect we want. The basic idea behind this is the same as the file and console IO; within the IO object is hidden the entire state of the world. By changing this object, we can cause changes in the state of the world (such as windows appearing). Assume, for a moment, that we do not want to change the *program* state, but just the IO state, by making a window appear. The *ChangeIOState* function takes a list of functions which change the IO state and applied each in turn.

6a ⟨*HelloWindow* 6a⟩≡ (9)

HelloWindow state io ≡ (*state*,*ChangeIOState* ⟨*create window* 6b⟩ *io*)

Assuming, of course, that we can write some code for ⟨*create window* 6b⟩ that will actually cause the window to pop up in the way we want. It should come as no surprise that we can, quite easily. The *OpenWindows* function takes a list of window definitions, and creates one new window for each.

6b ⟨*create window* 6b⟩≡ (6a)

[*OpenWindows* ⟨*list of windows to create* 6c⟩]

We only want to create a single window, so there is only one element in the list.

6c ⟨*list of windows to create* 6c⟩≡ (6b)

[*FixedWindow* 1 (0,0) **"Very␣Simple␣Application"**
((0,0),(200,50)) *DrawMessage*
[*StandByWindow*, *GoAway* *QuitCallback*]]

The *FixedWindow* function is used to create a single fixed (that is, non-scrollable) window. It has several arguments, which are quite straightforward:

- An ID number which is used to identify the window when operations need to be performed on it. Any integer can be used here, but each window should have a distinct number.
- A position which the window should appear at. The window will be opened at this point on the screen if possible.
- A window title, which will be displayed in the title bar of the window.
- A *PictureDomain*. The picture domain describes how large the window will be (200 pixels by 50 pixels in the example above).
- An update function, which will be used to draw the contents of the window. This is a function of a very specific type which will be called with appropriate arguments by the Clean runtime system when the window needs to be drawn. In the example above the function *DrawMessage* is used.
- A list of window attributes. There are several special attributes which can be given to windows. One of these is a "go away" function. This is a function which the Clean runtime will call if the "go away" or "close" button on the window is pressed.

2.4 Drawing in windows

The *DrawMessage* function mentioned above shall be used to place an encouraging and witty message in a window. As with the *HelloWindow* function, it works by adjusting the IO state of the program. This time, instead of using the entire IO state of the program, we only get what is known as an *UpdateArea*, which describes a region of a window which the function may draw in. We also have a program state, of course, but in this example, we are not going to change that state in any way. Look, now, at the type signature for the *DrawMessage* function:

6d ⟨*DrawMessage signature* 6d⟩≡ (9)

DrawMessage :: *UpdateArea State* → (*State*, [*DrawFunction*])

The *DrawFunction* type is defined in the clean IO library. It is the type of functions which will cause some change in an *UpdateArea*, such as adding graphical objects to a window. A function which adds a line of text to the *UpdateArea* is

6e ⟨*DrawMessage* 6e⟩≡ (9)

DrawMessage area state ≡ (*state*,[*MovePenTo* (10,40), *DrawString* **"Hello␣*World"**])

Each drawing window has a concept of a current pen position (this will be familiar to users of, for example, the apple QuickDraw package on which the Clean IO library is partially modelled). Many objects are drawn at the current pen position, so it is often necessary to move the pen before adding such objects. Text, added using the *DrawString* function, is an example of this.

2.5 Bringing it all together

As with all clean programs, we need a *Start* function to kick the program off. For IO programs, this function is often more complex than the standard *Start* for console programs, as we have to initialize the IO system, and create some initial objects. As described in the clean documentation, IO programs, also known as *World mode* programs, have the following type

7a ⟨*Start signature* 7a⟩≡ (9)

$$Start\ ::\ \star\mathbf{World}\ \rightarrow\ \star\mathbf{World}$$

The canonical form of such a function is

7b ⟨*Start* 7b⟩≡ (9)

$$\begin{aligned}&Start\ \ world\ \equiv\ CloseEvents\ \ events'\ \ world'\\&\qquad\qquad\mathbf{where}\\&\qquad\qquad\langle Start\ local\ definitions\ \text{7c}\rangle\end{aligned}$$

This may seem a little backwards at first glance – the program starts by closing the IO event stream! This is correct, of course. The program is evaluated by closing the *final* events (here called *events'*), which are produced by evaluating all of the users interactions with the program. We can generate these final events in the local definitions of the *Start* function.

We begin by opening the event stream. That is, we remove the event stream from the "world", like this:

7c ⟨*Start local definitions* 7c⟩≡ (7b) 7d ▷

$$(\,events,world'\,)\ \equiv\ OpenEvents\ \ world$$

Next, we generate the "final events" by processing the event stream.

7d ⟨*Start local definitions* 7c⟩+≡ (7b) ◁ 7c 7f ▷

$$(\,\_,events'\,)\ \equiv\ StartIO\ \ [\,menu\,]\ \ StartState\ \ [\,HelloWindow\,]\ \ events$$

The *StartIO* function generates a final state (here discarded) and a final set of events by processing the event stream. It also allows some initialization to be performed. This will typically be used to generate the initial user interface to interact with. We will create two objects in the user interface: A menu, to allow the user to quit the program, and a window containing our message. The menu can be created quite easily. We will need some numeric identifiers to keep track of the menu (actually, for this simple program, we don't need them, but it's good practice to use them anyway):

7e ⟨*Window system handles* 7e⟩≡ (9)

$$\begin{aligned}&MenuId\ \equiv\ 1\\&QuitItemId\ \equiv\ 10\end{aligned}$$

we can then create the menu quite simply:

7f ⟨*Start local definitions* 7c⟩+≡ (7b) ◁ 7d

$$\begin{aligned}&menu\ \equiv\ MenuSystem\ [\,PullDownMenu\ \ MenuId\ \ \text{``}\mathbf{File}\text{''}\ \ Able\ \ [\\&\qquad\qquad MenuItem\ \ QuitItemId\ \ \text{``}\mathbf{Quit}\text{''}\ \ (\,Key\ \ \text{`}\mathbf{Q}\text{'}\,)\ \ Able\ \ QuitCallback\,]\,]\end{aligned}$$

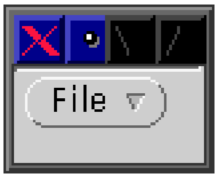

Figure 6: Menu window under UNIX.

How the menus appear depends on which version of the Clean IO system you are using. On a UNIX workstation, the menus appear in a free-floating window, as shown in figure 6.

2.6 Callbacks

We have not yet talked about *callbacks*. This concept will be familiar to anybody who done imperative programming using a GUI toolkit (such as the X toolkit intrinsics). The basic idea is that objects like menus and buttons have functions "attached" to them which are called when the objects are used. In our little example program, the "quit" item on the file menu has a function called *QuitCallback* attached to it. When the user selects this menu item, the function is called and can perform some action which will cause the program to exit. The clean IO library provides a function which can be used by our *QuitCallback* function to terminate the program. All callbacks in clean have the same type:

8a ⟨*Quit callback* 8a⟩≡ (9) 8b ▷

$QuitCallback :: State\ IO \to (State,\ IO)$

Our quit function is quite simple; we shall use the *QuitIO* function provided in the clean IO library to tell the *Start* function that there will be no more events to process:

8b ⟨*Quit callback* 8a⟩+≡ (9) ◁ 8a

$QuitCallback\ state\ io \equiv (state,\ QuitIO\ io)$

2.7 Program State

We shall define a type *State*, to store the program state. Now, this program doesn't actually *have* any state, but most interesting programs will, so we shall define this anyway, as an illustration of how it's done. Consider this to be a dummy, or stub. Note that we encode the uniqueness information in the type rather than in the type signature of the functions using it. We could do either, but this approach tends to look neater in large programs.

8c ⟨*State type* 8c⟩≡ (9)

$::\ \star IO \equiv IOState\ State$

$::\ \star State \equiv \textbf{Bool}$

We do need to initialize the program state, because of the structure of the *StartIO* function, so we provide a simple function to do so:

8d ⟨*Initial State* 8d⟩≡ (9)

$StartState :: \star State$

$StartState \equiv \textbf{True}$

We shall need only a single module for this program, which we shall call `Hello.icl`. Of course, this contains the usual preamble required by a clean program.

9 ⟨*Hello.icl* 9⟩≡

module *Hello*
⟨*library imports* 5a⟩
⟨*State type* 8c⟩
⟨*HelloWindow type* 5b⟩
⟨*HelloWindow* 6a⟩
⟨*DrawMessage signature* 6d⟩
⟨*DrawMessage* 6e⟩
⟨*Initial State* 8d⟩
⟨*Quit callback* 8a⟩
⟨*Window system handles* 7e⟩
⟨*Start signature* 7a⟩
⟨*Start* 7b⟩

2.8 Exercises

1. Modify the program so that it displays your own message instead of "Hello, *World".
2. Change the size of the window, and the position of the text within it.

3 Polychromatic greetings

Let's expand our "Hello World" example to allow the user to change the colours the message is displayed in, and even the message itself. This will serve as an example of how to write more complex menu callbacks, and how to create dialog boxes containing standard window controls such as buttons.

3.1 Basic structure

For this program, we need to remember what colours we want to draw the message in, and what the message will be. We shall keep the state in a record type. This is quite natural in clean, as it is possible to update a record object without the need to "dismantle" the entire thing (as one would have to do with, for example, a tuple). This is a standard way to declare the program state. You will probably have noticed similar code in the previous example. A single object of this type will be maintained as the program executes, and we will update it with all the information we wish to retain.

First, a handy *IO* object to use in the signatures of out functions.

10a ⟨*program state* 10a⟩≡ (19b) 10b ▷

$:: \star IO \; \equiv \; IOState \;\; State$

Uses `State` 10b.

IO And now the real state of the program. The *Colour* type is provided by the IO library (note the European spelling).

10b ⟨*program state* 10a⟩+≡ (19b) ◁ 10a 10c ▷

$$\begin{array}{l} :: \star State \; \equiv \; \{ \\ \qquad \begin{array}{lll} pen & :: & Colour, \\ back & :: & Colour, \\ message & :: & \textbf{String} \end{array} \\ \qquad \} \end{array}$$

Defines:
`State`, used in chunks 10, 12, 13, and 17–19.

Initially, we want the program to look like the previous example, so the initial state is set up to duplicate this. The constants *BlackColour* and *WhiteColour* are defined in the Clean IO library.

10c ⟨*program state* 10a⟩+≡ (19b) ◁ 10b

$StartState \; :: \; \star State$

$StartState \; \equiv \; \{pen{\equiv}BlackColour, \;\; back{\equiv}WhiteColour, \;\; message{\equiv}\text{“}\textbf{Hello,␣*World}\text{”}\}$

Defines:
`StartState`, used in chunk 10d.
Uses `State` 10b.

The overall form of the program is similar to the previous example. Note that this time we shall create the main window (called *msgwin*) in the same place as the menus (just to show the different style).

10d ⟨*Start* 10d⟩≡ (19b)

$$\begin{array}{l} Start \; :: \; \star\textbf{World} \; \rightarrow \; \star\textbf{World} \\ Start \;\; world \; \equiv \; CloseEvents \;\; events' \;\; world' \\ \qquad\qquad \textbf{where} \\ \qquad\qquad\quad (events,world') {\equiv} OpenEvents \;\; world \\ \qquad\qquad\quad (\_,events') {\equiv} StartIO \;\; [menu,msgwin] \;\; StartState \;\; [\,] \;\; events \\ \qquad\qquad\quad \langle \textit{menu definitions } 11a \rangle \\ \qquad\qquad\quad \langle \textit{message window definition } 11c \rangle \end{array}$$

Defines:
`Start`, never used.
Uses `StartState` 10c.

Figure 7: Menus for the ColourHello application.

The menus are not much more complex this time; we just add another menu with a single option which will create the dialog box which allows us to set the options. The menus look similar to those in the previous program (see figure 7).

11a ⟨*menu definitions* 11a⟩≡ (10d)

menu ≡ *MenuSystem* [*PullDownMenu* *FileID* **"File"** *Able* [
MenuItem *QuitID* **"Quit"** (*Key* **'Q'**) *Able* *QuitCB*],
PullDownMenu *OptionID* **"Options"** *Able* [
MenuItem *ShowID* **"Set␣options"** (*Key* **'S'**) *Able* *ShowCB*]]

Uses `QuitCB` 13a and `ShowCB` 13b.

We need the usual collection of handles to identify the menus we have created. As always, the numbers we pick are arbitrary, but it helps to follow some pattern so that they are unique. So far these handles have not been very interesting. Later on in this program, though, we will see how to use them. For now, we just need to define some arbitrary numbers.

11b ⟨*window system handles* 11b⟩≡ (19b) 11d ▷

FileID ≡ 10
QuitID ≡ 11
OptionID ≡ 20
ShowID ≡ 21

The function to create the message window is almost identical to the one used previously, but this time we use *WindowSystem* rather then *OpenWindow*, because this is part of the initial IO specification, rather than the initial actions. A subtle difference, but an important one. There is no particular reason to choose one approach over the other in this program. The difference is mainly one of types, and isn't something you should worry about now. The values used to create the window are essentially the same as before.

11c ⟨*message window definition* 11c⟩≡ (10d 19b)

msgwin ≡ *WindowSystem* [*FixedWindow* *MsgWinID* (0,0) **"Dialog␣demo"**
((0,0),*bounds*) *DrawMessage*
[*StandByWindow*, *GoAway* *QuitCB*]]

Uses `bounds` 12b, `DrawMessage` 12a, and `QuitCB` 13a.

And, as always, we must give the window an ID.

11d ⟨*window system handles* 11b⟩+≡ (19b) ◁ 11b 13d ▷

MsgWinID ≡ 30

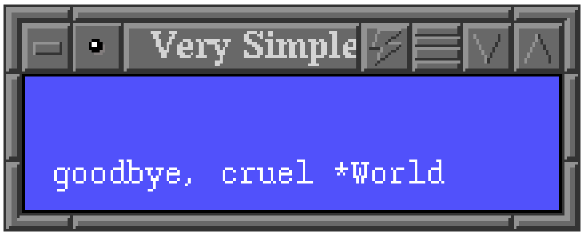

Figure 8: Main window for ColourHello application.

3.2 Colouring in the window

The program state will store the colours we will use to draw the message, as well as the message itself. Note the use of the *clear* function here; this will draw an empty rectangle as large as the window in the colours we want, so that the entire background of the window is the colour we want. The window will typically look like the one shown in figure 8. A *DrawFunction* is a function which places some drawing object in a window. We have already seen a simple example of this in the previous program. This function performs several operations which must be done in the right order. Luckily the list of drawing functions is sequenced, is we can be sure about what order things will happen in.

12a ⟨*show message in window* 12a⟩≡ (19b)

$$
\begin{array}{l}
\mathit{DrawMessage} :: \mathit{UpdateArea}\ \mathit{State} \rightarrow (\mathit{State},[\mathit{DrawFunction}])\\
\mathit{DrawMessage}\ \mathit{area}\ \mathit{state} \equiv (\mathit{state}',[\mathit{move},\mathit{setpen},\mathit{setback},\mathit{clear},\mathit{drawit}])\\
\qquad \textbf{where}\\
\qquad \mathit{move} \equiv \mathit{MovePenTo}\ (10,40)\\
\qquad \mathit{setpen} \equiv \mathit{SetPenColour}\ \mathit{state.pen}\\
\qquad \mathit{setback} \equiv \mathit{SetBackColour}\ \mathit{state.back}\\
\qquad \mathit{clear} \equiv \mathit{EraseRectangle}\ ((0,0),\mathit{bounds})\\
\qquad \mathit{drawit} \equiv \mathit{DrawString}\ \mathit{state.message}\\
\qquad \mathit{state}' \equiv \{\mathit{state}\&\mathit{message}{\equiv}\mathit{state.message}\}
\end{array}
$$

Defines:
`DrawMessage`, used in chunks 11c and 17e.
Uses `bounds` 12b and `State` 10b.

Notice how we produce *state*′. We don't actually want to change anything in the state (least of all the message), but the compiler doesn't like us just saying $\mathit{state}'{\equiv}\mathit{state}$, because we have inspected the contents of *state* several times in other clauses. To satisfy the compilers attempts to preserve the uniqueness of *state* we must make a bogus change. Don't let it bother you.

We shall use the size of the message window several times, so it's handy to have it as a separate named value. It really helps readability.

12b ⟨*message window size* 12b⟩≡ (19b)

$$\mathit{bounds} \equiv (200,50)$$

Defines:
`bounds`, used in chunks 11c, 12a, and 17e.

3.3 Callbacks

So, we have written the functions which will make the windows appear and draw what we want in it (based on the state). What now? We need to make the program properly interactive by writing code for the menu operations.

3.3.1 Menus

Quitting out of the program is as easy as it was in the first example. Unless a program needs to perform some special action on quitting (such as saving files) the quit function will tend to look like this. Remember, *QuitIO* will terminate the evaluation of *StartIO*.

13a ⟨*Quit menu option callback* 13a⟩≡ (19b)

$$QuitCB :: State\ IO \rightarrow (State, IO)$$
$$QuitCB\ s\ i \equiv (s, QuitIO\ i)$$

Defines:
`QuitCB`, used in chunk 11.
Uses `State` 10b.

When the "Set options" function is selected from the menu, this function is called. It will create a dialog box which can be used to set the colours, etc. in the program.

13b ⟨*show options dialog callback* 13b⟩≡ (19b)

$$ShowCB :: State\ IO \rightarrow (State, IO)$$
$$ShowCB\ s\ io \equiv (s', io')$$
$$\textbf{where}$$
$$io' \equiv OpenDialog\ theDialog\ io$$

⟨*dialog box specification* 13c⟩

Defines:
`ShowCB`, used in chunk 11a.
Uses `State` 10b.

3.3.2 Dialog box

This is the really important bit, so pay attention. There is a standard function in the Clean library called *OpenDialog*. It takes a dialog specification and transforms the *IO* state so that an appropriate dialog box appears on the screen. We will look in some detail at how to specify a dialog box, and how to place controls in it.

First, there is the dialog box itself. We don't have to specify a geometry for the box, as it will resize itself to fit all the controls we specify. We can set a list of options for the dialog box (here called *dlgOptions*), and specify a list of controls to be placed in the dialog box (*controls*). These controls will have callbacks attached to them which allow the dialog box to have effects on the program state. The final dialog box is shown in figure 9.

13c ⟨*dialog box specification* 13c⟩≡ (13b) 13e ▷

$$theDialog \equiv CommandDialog\ OptionDlgID\ \textbf{``Options''}\ dlgOptions\ 0\ controls$$

Of course, we need the usual ID numbers. Up till now the ID numbers we have been using have not been very interesting. Now, however, they become very important. The ID numbers are our handles into the dialog box which we will use to refer to various controls. If each control does not have a specific ID then we cannot fetch or set its value.

13d ⟨*window system handles* 11b⟩+≡ (19b) ◁ 11d 14d ▷

$$OptionDlgID \equiv 40$$

The dialog box can have a number of different options set (none of which are required). Here we choose to set the margin around the controls to six pixels.

13e ⟨*dialog box specification* 13c⟩+≡ (13b) ◁ 13c 14a ▷

$$dlgOptions \equiv [DialogMargin\ (Pixel\ 6)\ (Pixel\ 6)]$$

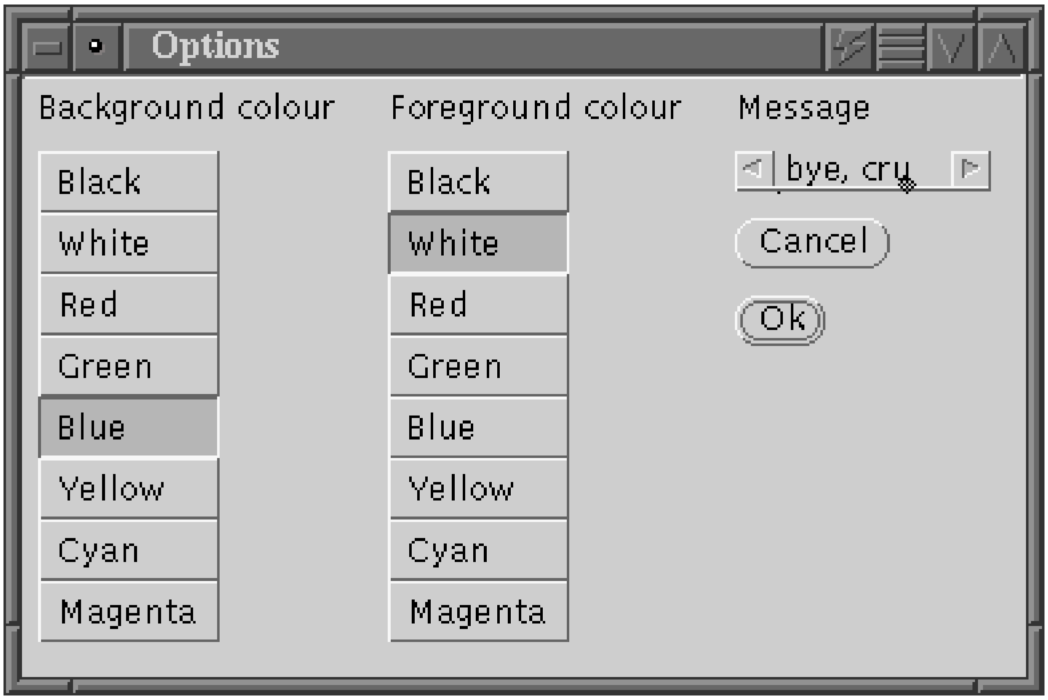

Figure 9: Options dialogue box for the ColourHello application.

We shall place eight controls in the window, two of which are complex controls (i.e. they contain other controls). We shall examine each one in turn. Now might be a good time to refer back to ⟨*dialog box specification* 13c⟩ where they are used.

14a ⟨*dialog box specification* 13c⟩+≡ (13b) ◁13e 14b▷

$controls \ \equiv \ [head1,\ head2,\ head3,\ bg,\ fg,\ messagecontrol,\ cancelbtn,\ okbtn]$

Finally, note that because we have used the program state in setting the defaults for the controls, we must create a new state to preserve the uniqueness of the state.

14b ⟨*dialog box specification* 13c⟩+≡ (13b) ◁14a 14c▷

$s' \ \equiv \ \{s\&pen{\equiv}s.pen\}$

3.3.3 Headings

The three "head" controls are just static bits of text which act as headings for the various parts of the dialog box. We give them ID numbers so that we can position the other controls relative to them. The *StaticText* widget contains a text label which cannot be edited by the user.

14c ⟨*dialog box specification* 13c⟩+≡ (13b) ◁14b 15a▷

$head1 \ \equiv \ StaticText\ \ H1ID\ \ Left$ **"Background␣colour"**
$head2 \ \equiv \ StaticText\ \ H2ID\ \ (XOffset\ \ H1ID\ \ (Pixel\ \ 20))$ **"Foreground␣colour"**
$head3 \ \equiv \ StaticText\ \ H3ID\ \ (XOffset\ \ H2ID\ \ (Pixel\ \ 20))$ **"Message"**

The Id numbers we use are

14d ⟨*window system handles* 11b⟩+≡ (19b) ◁13d 15b▷

$H1ID \ \equiv \ 41$
$H2ID \ \equiv \ 42$
$H3ID \ \equiv \ 43$

3.3.4 Headings

The "bg" (short for background) control contains a number of buttons, only one of which may be selected at any moment (like the buttons on an car radio). The things to note here are:

- The bank of radio buttons has an ID (*BGID*), which can be used to select it and find out which button is currently pressed. We will see how to do this shortly.
- We specify that the buttons should appear beneath the first heading in the dialog, by using *Below H1ID* as its position indicator.
- We would like the buttons laid out as a single column of buttons, which is specified by saying *Columns* 1. We may specify either a number of rows or a number of columns, but not both (the number of buttons is used to calculate the other figure).
- When the buttons are created, one of them must be selected (as a default). We select the one called *sbg*, which is actually a function which calculates the correct button based on the program state.

We finish the specification with a list of *RadioItem*s, here called *bgbuttons*. The colours available on these buttons are not random. These are the standard colours available in the Clean IO library. Any other colours must be specified with RGB values. The *Select* function is the callback for each of these buttons. Don't worry about what it does (actually, it doesn't really do anything).

15a ⟨*dialog box specification* 13c⟩+≡ (13b) ◁14c 15c▷

```
bg ≡ RadioButtons BGID (Below H1ID) (Columns 1) sbg bgbuttons
bgbuttons ≡
        [
        RadioItem BGBlackID "Black" Able Select,
        RadioItem BGWhiteID "White" Able Select,
        RadioItem BGRedID "Red" Able Select,
        RadioItem BGGreenID "Green" Able Select,
        RadioItem BGBlueID "Blue" Able Select,
        RadioItem BGYellowID "Yellow" Able Select,
        RadioItem BGCyanID "Cyan" Able Select,
        RadioItem BGMagentaID "Magenta" Able Select
        ]
```

Uses `Select` 19a.

Each control needs a unique ID. Note the pattern used in assigning ID numbers to the buttons in the radio bank. The *RadioButtons* container is numbered "50", and each button within the container has ID number "50+colour number" where the colour number is arbitrary, but will be the same for the foreground colours. This consistency will shortly be used to make our lives easier.

15b ⟨*window system handles* 11b⟩+≡ (19b) ◁14d 16c▷

```
BGID ≡ 50
BGBlackID ≡ 51
BGWhiteID ≡ 52
BGRedID ≡ 53
BGGreenID ≡ 54
BGBlueID ≡ 55
BGYellowID ≡ 56
BGCyanID ≡ 57
BGMagentaID ≡ 58
```

The radio buttons for selecting the foreground are very similar to the ones for the background. Let's create these in a different way to the background buttons. Firstly, we want some function *fg* which is essentially identical to the *bg* function:

15c ⟨*dialog box specification* 13c⟩+≡ (13b) ◁15a 16a▷

```
fg ≡ RadioButtons FGID (Below H2ID) (Columns 1) sfg fgbuttons
```

Now, what was distinctive about the function which generated the background buttons? The amount of similarity between all of the declarations? That's right. Each of the declarations is a constant value, and each one has a similar form. If we could write some function to generate these values then we might be able to write simpler and clearer code. How about this?

16a ⟨*dialog box specification* 13c⟩+≡ (13b) ◁15c 16b▷

$makeButton\ id\ (no,\ nam) \equiv RadioItem\ (id{+}no)\ nam\ Able\ Select$

Uses `Select` 19a.

In this function, the *id* value is the ID number of the bank of radio buttons (remember, it's *BGID* for this example), *no* is the number *within the radio bank* of this button (so that *id*+*no* is the ID number of this button), and *nam* is a string which will be the text shown on the button.

Now this function will not be very useful if we have to list all of the ID values and strings anyway. Well, this is where the power of functional programming comes to the rescue! We can use some of the standard higher-order functions to construct the pairs we need (that's why we used pairs in the *makeButton* function above). Note how the ID number of the button is constructed. We are following the same pattern we used to construct the background button ID numbers, but this time all of the numbers are unnamed. This means that if we want to refer to a specific button we will have to calculate it's ID number. On the other hand, this saves us naming lots of constants which we may never use. In this program, in fact, we will always be calculating the ID numbers.

16b ⟨*dialog box specification* 13c⟩+≡ (13b) ◁16a 16d▷

$fgbuttons \equiv map\ (makeButton\ FGID)$
$\quad (zip\ ([1\ldots], [\text{“Black”}, \text{“White”}, \text{“Red”}, \text{“Green”}, \text{“Blue”}, \text{“Yellow”}, \text{“Cyan”}, \text{“Magenta”}]))$

The only ID number we need to assign is this:

16c ⟨*window system handles* 11b⟩+≡ (19b) ◁15b 16f▷

$FGID \equiv 60$

The functions to select the defaults in the radio banks based on the program state simply convert the currently selected colour to the appropriate ID number. This was the point of organising the ID numbers the way we did.

16d ⟨*dialog box specification* 13c⟩+≡ (13b) ◁16b 16e▷

$sfg \equiv (ColToIdOffset\ s.pen){+}FGID$
$sbg \equiv (ColToIdOffset\ s.back){+}BGID$

Uses `ColToIdOffset` 18b.

3.3.5 The Text Control

The control to allow the user to type in a new message is called an *EditText* control. It's specification is quite simple. We say where it appears, it's width (in pixels), and the number of lines of text which are allowed. We also supply a default piece of text to appear in the control when it is created, which is just the current message as taken from the program state.

16e ⟨*dialog box specification* 13c⟩+≡ (13b) ◁16d 17a▷

$messagecontrol \equiv EditText\ MsgID\ (Below\ H3ID)\ (Pixel\ 100)\ 2\ s.message$

Of course, like everything else, we need an ID number so that we can refer to it later

16f ⟨*window system handles* 11b⟩+≡ (19b) ◁16c 17c▷

$MsgID \equiv 70$

3.3.6 Buttons

It's a good idea to place a "cancel" button in a dialog box, which will undo any changes selected in the box. The *CancelChange* callback will make the dialog box vanish, and all the changes will be undone. This is an example of a *DialogButton* control, which is just a standard button widget with a piece of text in it.

17a $\langle$*dialog box specification* 13c$\rangle$+≡ (13b) ◃16e 17b▹

$cancelbtn \equiv DialogButton\ CancelID\ (Below\ MsgID)$ **"Cancel"** $Able\ CancelChange$

Uses `CancelChange` 17d.

The "OK" button applies all the changes selected in the dialog, and makes it vanish. The *RedrawMessage* callback is the function which updates the program state and makes the dialog disappear.

17b $\langle$*dialog box specification* 13c$\rangle$+≡ (13b) ◃17a

$okbtn \equiv DialogButton\ OkID\ (Below\ CancelID)$ **"Ok"** $Able\ RedrawMessage$

Uses `RedrawMessage` 17e.

These buttons will need ID numbers as well

17c $\langle$*window system handles* 11b$\rangle$+≡ (19b) ◃16f

$OkID \equiv 44$
$CancelID \equiv 45$

3.3.7 Dialog box callbacks

When the cancel button is pressed, all we need to do is pop down the dialog box without changing the program state. The *CloseActiveDialog* function will do this for us.

17d $\langle$*callbacks* 17d$\rangle$≡ (19b) 17e▹

$CancelChange :: DialogInfo\ State\ IO \rightarrow (State,IO)$
$CancelChange\ dialog\ s\ io \equiv (s,io')$
where
$io' \equiv CloseActiveDialog\ io$

Defines:
`CancelChange`, used in chunk 17a.
Uses `State` 10b.

When the "Ok" button is pressed, we need to update the program state to reflect the changes made in the dialog, and redraw the message window so that the changes take effect. We use the *updateState* function to make the changes to the program state based on the current state of the dialog box. Not only that, but we call the drawing function we defined earlier on, using the new state, so that the changes are shown in the main window.

17e $\langle$*callbacks* 17d$\rangle$+≡ (19b) ◃17d

$RedrawMessage :: DialogInfo\ State\ IO \rightarrow (State,IO)$
$RedrawMessage\ dialog\ s\ io$
$\equiv (s'',redraw)$
where
$closeDlgIo \equiv CloseActiveDialog\ io$
$(s'',drawit) \equiv DrawMessage\ [((0,0),bounds)]\ s'$
$redraw \equiv DrawInActiveWindow\ drawit\ closeDlgIo$
$s' \equiv updateState\ dialog\ s$

Defines:
`RedrawMessage`, used in chunk 17b.
Uses `bounds` 12b, `DrawMessage` 12a, `State` 10b, and `updateState` 18a.

3.4 Support functions

We need to convert the control ID's supplied in the *DialogInfo* into actual colours. A couple of handy functions do this for us. Note how we can use the control query functions *GetSelectedRadioItem* and *GetEditText* to determine the status of the controls in the dialog. This is where our care in choosing ID numbers for our radio button controls will pay off.

18a ⟨*miscellaneous functions* 18a⟩≡ (19b) 18b ▷

```
updateState :: DialogInfo State → State
updateState dlg s ≡ { pen ≡ penCol, back ≡ backCol, message ≡ msg }
                    where
                     penCol ≡ IdOffsetToCol ((GetSelectedRadioItemId FGID dlg)−FGID)
                     backCol ≡ IdOffsetToCol ((GetSelectedRadioItemId BGID dlg)−BGID)
                     msg ≡ GetEditText MsgID dlg
```

Defines:
`updateState`, used in chunk 17e.
Uses `IdOffsetToCol` 18b and `State` 10b.

Converting from Colours to ID numbers and back again is pretty simple, because our ID numbers follow a simple pattern. The *abort* function will terminate the program immediately, and print its string argument out as an error message.

18b ⟨*miscellaneous functions* 18a⟩+≡ (19b) ◁ 18a

```
ColToIdOffset :: Colour → Z
ColToIdOffset c ≡ case c of
                   BlackColour → 1
                   WhiteColour → 2
                   RedColour → 3
                   GreenColour → 4
                   BlueColour → 5
                   YellowColour → 6
                   CyanColour → 7
                   MagentaColour → 8
                   _ → abort "Unknown␣Colour!\n"

IdOffsetToCol :: Z → Colour
IdOffsetToCol id ≡ case id of
                   1 → BlackColour
                   2 → WhiteColour
                   3 → RedColour
                   4 → GreenColour
                   5 → BlueColour
                   6 → YellowColour
                   7 → CyanColour
                   8 → MagentaColour
                   _ → abort ("Unknown␣widget␣ID\n"+(toString id))
```

Defines:
`ColToIdOffset`, used in chunk 16d.
`IdOffsetToCol`, used in chunk 18a.

The callback used when the user presses a button is actually a dummy callback, because nothing happened when the button is pressed. It's only when the "Ok" button is pressed that things actually start to happen.

19a ⟨*Radiobutton selection* 19a⟩≡ (19b)

$Select :: DialogInfo\ (DialogState\ State\ IO) \rightarrow DialogState\ State\ IO$
$Select\ dinfo\ dstate \equiv dstate$

Defines:
`Select`, used in chunks 15a and 16a.
Uses `State` 10b.

3.5 Assembling the final program

The chunks go together like this to make the entire program.

19b ⟨*ColourHello.icl* 19b⟩≡

module *ColourHello*
import *StdEnv*, *deltaEventIO*, *deltaIOSystem*, *deltaWindow*
import *deltaSystem*, *deltaPicture*, *deltaFont*, *deltaDialog*
⟨*Radiobutton selection* 19a⟩
⟨*show options dialog callback* 13b⟩
⟨*message window size* 12b⟩
⟨*Start* 10d⟩
⟨*window system handles* 11b⟩
⟨*Quit menu option callback* 13a⟩
⟨*program state* 10a⟩
⟨*show message in window* 12a⟩
⟨*message window definition* 11c⟩
⟨*callbacks* 17d⟩
⟨*miscellaneous functions* 18a⟩

3.6 Exercises

1. Add an "apply" button to the dialog box which will cause the new selections to take effect, but will not close the dialog box.
2. Extend the "apply" button so that if the "cancel" button is selected, the changes will be undone (i.e. have the "apply" button allow previewing).
3. Add the ability to select a font to the dialog box, and have the text be displayed in this font (difficult).

At the end of this first tutorial, you should

1. Be comfortable with the concepts of IO programming in Clean. This includes unique types, and how they are used to manage the state of a program, and the general form of a World mode Clean program.
2. Be comfortable enough with the Clean IO library to write small applications of your own, using the functions you have been shown.

Once you understand the basic material presented in this tutorial, you will be ready to tackle some of the more advanced concepts in the Clean IO library (mouse clicks, drawing graphics and file operations).

A How to read this document

This document has been written using a style known as "literate programming". This section serves as a basic introduction to literate programming, to allow the reader to understand a program written in this style.

The central idea in literate programming is "writing programs for people to read", as opposed to writing programs for machines to read (via compilers).

In a literate program, documentation and description are intermingled with program code, in whatever order the author feels will allow the reader to understand the program. When the program is to be compiled, a processor (usually called `tangle`) is used to extract the program source code, and rearrange it into the correct order. When a printed document is required, a processor (usually called `weave`) is used to generate it. This document has been prepared using `weave` and the TeX document processor.

When `weave` prepares TeX input from a web (the source for a literate program is called a web), certain things are added to the document to facilitate understanding of the program. Once these additions are understood, they greatly facilitate the easy understanding of the program under consideration. The remainder of this section discusses these additions. Note that there are many web systems, and each one produces slightly different output. The system described here is called `noweb`.

A.1 Navigation Information

Each code chunk is accompanied by several pieces of navigation information to help the reader understand how the chunks fit together to make a complete program.

Each chunk has a name, which is contained within angle brackets at the start of the chunk. If this is the first definition of the chunk, then it is followed by a $\equiv$ symbol. If this chunk is a continuation of a previous chunk, then the symbol is $+\equiv$. In the margin to the left of this chunk name is a reference symbol for the chunk, based on the page number which the chunk appears on (for instance, the second chunk on page six is known as *6b*. These numbers appear wherever a chunk is referred to, and allow the reader to quickly locate the definitions of chunks which are referenced. In the right hand margin several pieces of information appear. Within bracket is the location of the *use* of this chunk, which tells the reader where to find a chunk which uses this chunk. If this chunk is a continuation of a previous chunk, the location of the previous chunk appears, preceded by a left arrow. If the chunk is continued later on, then the next appearence of this chunk also appears, followed by a right arrow.

Within the body of the chunk, other chunks can be referenced. When the program code is produced (for the compiler), each reference is relpaced by the contents of that chunk. Each reference is accompanied by the page location of the chunk being used.

Following each chunk there may be *identifier information*, which tells the reader about any important identifiers which are defined in this chunk, and any identifiers which are used in the chunk. Not every identifier is listed in this cross reference information; the writer selects identifers of particular importance to be cross referenced in this section. Each identifier is followed by a reference to the chunks in which it is defined (or used, if it is defined in this chunk) to allow the reader to follow the use of identifiers throughout the program.

All of this cross reference information can be ignored by the reader, although chunk continuations and uses should be examined to expose the structure of the program.

A.2 Prettyprinting

The code in this document has been treated by a tool called a `prettyprinter` as it was woven. This tool alters the printed appearance of the code to improve its readability. In general, it is not difficult to understand the program code, as most of the prettyprinting simply consists of selecting appropriate fonts for different parts of the program, and indenting it correctly. Certain symbols, however, are printed in a very different form to the `ASCII` form which is required by the compiler. Greek letters, for instance, are used in place of the appropriate identifiers (e.g. an identifier `alpha` is printed as α). Certain operators and other symbols are also replaced by symbols which are common in mathematics. This table describes the transformations which might cause some confusion. Once the transformations are understood, the program becomes clearer.

ASCII symbol	Typeset symbol	Comments
`=`	$\equiv$	
`=>`	$\equiv$	
`..`	$\ldots$	As seen in lists like `[1..10]`
`\`	λ	Inline lambda expressions
`*`	$\star$	Uniqueness information in type signature
`*`	$\times$	Multiplication
`<>`	$\neq$	
`<=`	$\leq$	
`>=`	$\geq$	
`~`	$-$	Unary minus
`/`	$\div$	Division
`^`	$\uparrow$	Exponentiation
`==`	$=$	Test for equality
`++`	$+\!\!+$	List concatenation
`+++`	$+\!\!+$	String concatenation
`not`	$\neg$	
`||`	$\vee$	Boolean "or"
`&&`	$\wedge$	Boolean "and"