Using ML for Windows Programming

Panagiotis Argiris & Chris M.P. Reade
{Panagiotis.Argiris,Chris.Reade}@brunel.ac.uk

http://www.brunel.ac.uk/~csstcmr

Computer Science Department
Brunel University, Uxbridge
England UB8 3PH

Contact person: Dr CM.P Reade

Abstract

Modern functional languages such as ML have the potential to simplify GUI programming by providing convenient levels of abstraction. The paper discusses current approaches to GUI programming and presents a new high level declarative toolkit for the construction of Windows programs in Standard ML. Examples are given to illustrate the toolkit and the potential benefits are discussed.

1. Introduction

Relatively recently, there has been a breakthrough in the application of functional languages to graphical user interface (GUI) programming [1,2,3,4]. Some of the ideas coming from this recent research have been used in the construction of our graphical user interface toolkit Visual ML for use with Windows 95 and Windows NT. The toolkit was designed to make Windows
programming easy allowing GUI construction to be done with a conventional functional programming style.

1.1 Current approaches to GUI programming

Human-computer interfaces are difficult to design and implement [5] and one of the reasons is that today’s programming languages [6] do not provide direct support for GUI construction. Current approaches to GUI programming make use of Procedural, Object Oriented, Visual languages as well as UI builders to address the UI portion of the software. Most procedural approaches adopt the event loop/callback model which is simple but imposes a specific programming style. On the other hand procedural languages like C are not always easy to use. Object Oriented languages like C++, provide very good facilities for GUI programming but they usually have complex semantics, poor type checking and do not mix well with higher order abstraction. Visual languages like LabView and Prograph offer new non-textual ways for expressing algorithms but they are not yet well established to be widely used. UI builders like Visual Basic and XDesigner are powerful tools used to automate the interface(layout) creation, however linking the behaviour of the interface(callbacks) to its geometry(layout) usually requires advanced programming skills.

1.2 A declarative approach using Standard ML

The polymorphic functional language Standard ML [7] offers excellent solutions to GUI construction. ML programs are exceptionally concise, with a friendly syntax, easier to reason about compared to imperative counterparts and offer interesting opportunities for parallel execution. The main characteristics of ML [8] which apply very well to GUI programming are:

- **Abstraction and Composition**

  GUI designs are inherently complex and reusable components, abstractions and construction tools are generally needed to manage the complexity. ML provides the ability to combine and define abstractions easily, using higher order functions, data objects with functional components and parameterised modules.
• **Typechecking and Polymorphism**

For practical ML programming, the freedom to abstract is tempered with a safety net provided by the flexible ML type system. This prevents abstractions from being mis-applied and thus ensures that what gets built makes reasonable sense. A finer control of safety is provided by the ability to easily define new types to suit new situations, including higher order and recursive types.

• **Declarativeness**

We use this term here to capture the degree of separation between the description of behaviour and the act of invoking behaviour which is a non-declarative side-effect. This can be important for simplicity in reusing and sharing designs and of course for referential transparency.

• **Prototyping**

Functional Programming has often been advocated for fast prototyping and after the recent work in functional GUIs, it is now adequate for UI prototyping also, a major concern in most prototyping activity today.

• **Supporting alternative models**

External interfaces usually impose specific models for running GUI programs. The implementation of different high level models is complex and needs to be further explored. Such exploration is well supported within a functional framework which allows relatively fast prototyping.

2. **An overview of the toolkit**

Our ML toolkit provides the ability to hide and abstract from the procedural event loop mechanism used by Windows [9] while it offers the same functionality in a declarative style. The toolkit uses a type checked access to the low level imperative world and supports many features some of which are:- an open style of representation and connection between levels; good compositionality and easy reuse of components; type checked communication between components and construction of components; the ability to separate concerns of geometry & logical behaviour when designing and finally distributed state for improving modularity.

Using the toolkit, the programmer constructs values (data objects) which represents GUI components or complete applications and then *runs* these values to begin the side-affecting
behaviour (window creation etc). The emphasis in programming is therefore on constructing values representing the GUI's appearance and behaviour.

The toolkit consists of functions for creating and combining components and a GUI program has the form of a network of reactive devices or gadgets which communicate by sending typed message values to each other. A clear distinction is made between the communication network which reflects the logical structure of an application and the hierarchical structure reflecting the geometry of the visible windows as illustrated in Figures 2 and 3 of Section 3.

2.1 Components of a GUI program

Components are data objects - in fact functions. The arguments for these functions are used to specialise a component (for example with attributes), to attach behaviours (which are also functions) and to instantiate connections for communication. This means that construction of an interface is (almost entirely) just a matter of applying functions to appropriate arguments, starting from a fixed set of primitive functions.

For example, assuming \( p \) is an integer connection port (described later), we can construct a button labelled “Push Me” (of size 50 by 20) which sends a fixed integer (5, say) to the port \( p \) whenever the button is pressed:-

\[
\text{val btn = simpleButton (50,20) \ "Push Me" (send (outport p) 5)}
\]

This defines a data object (\( \text{btn} \) of type \( \text{gadget} \)) but does not cause a button to appear or visible side-effects to happen. Such a gadget would normally be combined with other gadgets by function application and then a final gadget or application can be run with \text{startDemoWin} or \text{runApplication}. These latter functions elaborate the construction and initiate the side-effects so that a graphical interface appears and reacts to user events. The function:-

\[
\text{simpleButton : (Width*Height) -> string -> command -> gadget}
\]

has arguments for the button size, label and finally the behaviour, which in this case is a data object of type command. Typical commands are communications (via ports) to other components, and some basic operating-system commands. The basic polymorphic primitive which ensures type checked communication is the function \( \text{send} \). The types of the functions:-
send: ‘a outport -> ‘a -> command
outport: ‘a port -> ‘a outport

mean that the expression \texttt{send (outport p) 5} only type checks if \texttt{p: int port} because the value
being sent is \texttt{5:int}. Note that \texttt{btn} is really a connected component, because it has a fixed port to
communicate with. If we had wanted to make an unconnected component which could be
duplicated and re-used with different connections, we should make the output port a parameter
and define a function. For example:-

\[
\text{datatype click} = \text{Click} \\
\text{fun pushB outp} = \text{simpleButton (50,20) “Push Me” (send outp Click)}
\]

So \texttt{pushB: click outport -> gadget}, where \texttt{click} is a type whose only value is \texttt{Click} and \texttt{pushB}
can be applied to an appropriately typed outport to connect it. Similarly, a simple component
which displays strings which are sent to it from other components will be formed by applying
the function:-

\[
\text{stringDisplay : string -> string inport -> gadget}
\]
to arguments which describe the initial string and an inport connection for receiving new string
values to display.

There are several other primitive component functions: \texttt{winGadget} for building general
windows, \texttt{scrollbarGadget, editGadget, listboxGadget, comboboxGadget, staticGadget} for
common components as well as functions for describing dialogues, menus, abstract state
machines and timers. More details of these can be found in [10].

\section*{2.2 The Connection Problem}

It is not difficult in a polymorphically typed functional language to describe a particular network
with typed connections, nor is it difficult to provide generic means for constructing networks
(graphs) with a single type for all connections. Unfortunately, a generic means of constructing
networks with differently typed connections hits one of the limitations of (ML style)
polymorphism and there is no simple direct solution. An indirect solution adopted here, using
ports, is a variation on \texttt{wires} as described in [11] where a similar concept of \texttt{gadget} is introduced
for a purely applicative language. Ports act as typed connection points so that connecting up is
merely a matter of applying component functions to ports. However ports have identity so that
making two ports is not the same as sharing via two copies of the same port. This means that declaring ports causes side-effects which interfere with a purely functional view (undermining referential transparency). Such ‘uniqueness’ problems are well understood in purely functional languages and monads [12], continuations and uniqueness typing are some of the techniques used to solve them. These techniques either require an extension to the type system or a somewhat contorted programming style (or both). In the spirit of ML, we allow the side-effects resulting from declaring ports and provided certain precautions are taken, the ‘impurity’ can be isolated to a small part of the construction, so that most of the program can follow a clean functional style.

To illustrate the style of creating compound components, we assume we have a component which makes an abstract gadget (with a behaviour but no visual element):

\[
\text{counter: int -> string outport -> click inport -> gadget}
\]

The arguments to this function describe an initial state (an integer) a port for sending out string values and a port for receiving click values. The behaviour is to increment the state whenever a click is received and to send the string representation of this integer to the outport.

We wish to connect up such a component with a \texttt{(pushB)} button and a string display to make a self contained (complete) component. Given a start state (an integer), we construct three gadgets, and then use a layout function to place them in a row, treating the whole as a single gadget produced by \texttt{countGadget: int -> gadget}.

```plaintext
fun countGadget n
  = let val p1 = newport()
      and p2 = newport()
      val b = pushB (outport p1)
      val c = counter n (outport p2) (inport p1)
      val d = stringDisplay (stringofint n) (inport p2)
    in rowGadget[b,c,d] end
```

In contrast to the network connection problem, the organisation of the visual elements into a hierarchy is a simple matter of combining gadgets (connected components) with appropriate functions (like \texttt{rowGadget: gadget list -> gadget}).
2.3 Starting Gadgets and Applications

A final gadget can be run using \texttt{startDemoWin} which takes a title, a position and a gadget as arguments and causes the gadget to be started within a simple top level window:

\texttt{startDemoWin "count example" (100,100) (countgadget 0);}

For more sophisticated applications, we would use \texttt{runApplication} which supplies a quit command to the application so that it can be incorporated into a gadget’s behaviour. This means that an application is in fact a function which produces a gadget when supplied with a command argument (the quit command for the application).

\begin{verbatim}
val runApplication : application -> position -> unit
\end{verbatim}

\texttt{runApplication} supplies a \texttt{quit} command to the application so that it can be incorporated into a gadget’s behaviour. This means that an application is in fact a function which produces a gadget when supplied with a command argument (the quit command for the application).

\begin{verbatim}
type application = command -> gadget
\end{verbatim}

The programmer can incorporate a quit command in describing a gadget to be created by using a function:-

\begin{verbatim}
fun myApplication quit = (* construction of a gadget which can use quit - e.g in a button behaviour or menu *)
\end{verbatim}

The actual quit command is supplied internally by \texttt{runApplication}. The resulting gadget is then started and if the supplied quit command is executed when the gadget is running, it causes the running gadget to shut down cleanly. This allows applications to be started from an interactive ML console window, which is very convenient for experimentation.

In addition to \texttt{runApplication} and \texttt{startDemoWin} there are other primitives to start modal dialogues (either from the ML interactive top level or in response to an interaction in a running application). All these primitives automatically take care of forking a new process, creating and using message queues and providing a means for closing down applications cleanly. There is also a function:-

\begin{verbatim}
forkedGadget: gadget -> gadget
\end{verbatim}

which can be used in gadget construction and which causes a separate process to be forked when the resulting gadget is started. This means that the GUI programmer has some control over concurrency in an application without needing to see the details of how this is achieved.
A large variety of components can be described using basic gadget creation functions which take a list of optional attributes. For example, the possible attributes for windows are given by the type constructor in Figure 1 (\texttt{\texttt{\texttt{s}} \texttt{winAttrib}}) where the type parameter (\texttt{\texttt{s}}) is a type for a local state associated with the window.

Thus a single data structure describing many features of a window can be constructed and passed to a basic \texttt{winGadget} function. Note that several of the attributes involve functional values which represent some aspect of a window's behaviour. One of the major advantages of a functional language is that functions can be incorporated into data-structures in a simple way to allow complex descriptions to be built up. The type system is used extensively to capture exactly what form a description needs to take when describing basic components. Usually a behaviour function has a type of the form:-

\texttt{window} -> \texttt{'m} * \texttt{'s} -> \texttt{'s action}

where the parameter of type \texttt{window} allows reference to be made to the window object to which the behaviour is attached (and hence used in actions affecting the window); \texttt{'m} is the type of values or messages being sent to the gadget which the behaviour function responds to and \texttt{'s} is the type of a local state (of the gadget). Communications and other possible effects are represented as values of type \texttt{'a action} where \texttt{'a} is the type of the final value returned after the action has been performed. Behaviour functions produce an \texttt{'s action} value which is a data object describing effects and returning a new value (of type \texttt{'s}) for the new local state. So behaviour functions are like \texttt{next state} functions of a state machine and do not, by themselves, \texttt{execute} the actions. They only describe (declaratively) how a component should behave in response to receiving messages. The execution of the effects of actions is taken care of automatically when a gadget is started.
Behaviour functions are a convenient way of abstracting away from most of the low level details of callback functions which are used at the lower level interface to the operating system, whilst still encouraging use of distributed reactive components with separate states. The type constructor 'a action is a standard functional mechanism (a monad [12]) allowing us to manipulate values which have an effect only when executed and controlling how they are sequenced (using seqA, thenA and returnA). In particular, command is just an abbreviation for unit action. The counter component used above is defined as follows:-

fun counter n outp inp = let fun bhvr (Click,m) = send outp (stringofint (m+1)) seqA returnA (m+1) in abstractGadget bhvr n inp end

<table>
<thead>
<tr>
<th>datatype 's winAttrib</th>
</tr>
</thead>
<tbody>
<tr>
<td>= WinMenuBar of 's menuItem list</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 1 : Attributes for Windows
The behaviour function ($bhvr$) supplied to $\text{abstractGadget}$ does not use a window parameter as abstract gadgets do not have windows.

3. The game of life in Visual ML

For a more realistic example illustrating some of these features, we consider the game of life application whose visual appearance and component structure are depicted in Figures 2 and 3 respectively. This has a main window with a menu and side scrollbars and an active drawing area as well as a separate control window with a speed scrollbar control, a display and a few push buttons. The drawing area can be edited using the mouse to add or delete cells and the step and run buttons can be used to have new generations calculated and displayed. In the latter case, new generations are displayed at time intervals determined by a repeater component which can be adjusted.
In Figure 4, we show (most of) the definition of the life application \texttt{lifeApp}. In the body of this function, two datatypes are declared for messages (sent to a board and repeater components), and ports are declared to connect up the components.

This is followed by definitions of some buttons and an abstract gadget which is built with the primitive:

\texttt{repeaterGadget: \texttt{command} $\rightarrow$ \texttt{repeaterControl inport} $\rightarrow$ \texttt{gadget}}

making a repeater gadget where the argument of type \texttt{command} describes an effect which will happen at repeated intervals. In this case the effect is to send a value to a port.
fun lifeApp quit = 
let
  datatype boardaction = Resize of int | Step | Clear
  datatype repeaterControl = RepeaterStart | RepeaterStop
                   | RepeaterSpeed of int

  val p1 = newport() and p2 = newport()
  val toBoard = outport p1 and inBoard = inport p1
  and toRepeater = outport p2 and inRepeater = inport p2

  fun button s outp v = simpleButton (50,20) s (send outp v)

  val stopButton = button "Stop" toRepeater RptrStop
  val runButton = button "Run" toRepeater RptrStart
  val stepButton = button "Step" toBoard Step
  val clearButton = button "Clear" toBoard Clear
  val quitButton = simpleButton (50,20) "Quit" quit
  val repeater = repeaterGadget (send toBoard Step) inRepeater

  fun changeSize n _ s = send toBoard (Resize n) seqA returnA s

  val sizemenu = Popup ("Size",
    [ MenuItem ("Size 2x2", [MenuFunction (changeSize 2)]),
      MenuItem ("Size 4x4", [MenuFunction (changeSize 4)]),
      MenuItem ("Size 8x8", [MenuFunction (changeSize 8)]),
      MenuItem ("Size 12x12", [MenuFunction (changeSize 12)]) ]
  )

  val (speedDisplay, speedBar) = (* defines the speed controls *)

  val boardwin = 
    let val atrbs = [WinName "Life Demo",
                  WinMenuBar [sizemenu], (* other attributes *) ]
    in
      winGadget atrbs startState noChild
      boardFun inBoard
    end

  val controlwin = 
    let val atrbs = (* defines attributes for the control window *)
      val buttons = columnGadget[runButton,stopButton,stepButton,
                              clearButton, quitButton]
      val child = columnGadget [speedDisp, 
                                rowGadget[buttons,speedBar]]
    in
      winGadgetBasic atrbs () child
    end

in
  rowGadget[controlwin, boardwin]
end

Figure 4 : Definition of the Life Application
The size menu allows rescaling of the cell size using a simple behaviour function for menu items (\texttt{changeSize}). The main window gadget (\texttt{boardwin}) uses several behaviour functions (including \texttt{boardFun} which responds to board messages), whose definitions we omit. A separate control window (\texttt{controlwin}) contains the other components (we have also omitted definitions of \texttt{speedDisp}, \texttt{speedBar}, and the attributes). The whole application is simply started with:

\begin{verbatim}
runApplication lifeApp (30,40)
\end{verbatim}

4. Related work

There has been much recent work on functional graphical user interfaces. A number of functional toolkits currently exist which make use of the facilities provided by functional languages \cite{1,2,3,4,12,13}. Some approaches interface directly to the procedural event loop model as in \cite{1,2,13} while others make explicit use of concurrency \cite{3,4}.

The lazy language implementations (Fudgets \cite{2} for lazy ML; Clean \cite{1} and Haggis \cite{3} using Monadic IO in Haskell \cite{12}) differ in the way side-effects are expressed and also differ in the way composition is done. Our use of data structures with behaviour functions is similar to the Clean style, but we use message passing instead of sharing access to a global state (and outside world).

Our implementation in ML differs from the other declarative GUI toolkits in that it provides a declarative view of a large existing procedural Windows interface rather than building a new interface from scratch.

The non-lazy language implementations: eXene \cite{4} for New Jersey ML and the X and Motif interfaces in Poly/ML \cite{13} use a functional style (essentially higher-order procedures with side effects) and concurrency to create reusable abstractions for GUI programming. Our design has some things in common with the Poly/ML for X and Motif model, where behaviour functions are attached to widgets passing messages. However, our construction is made to appear more declarative because attachment is done by function application and the communication network is made more explicit with ports. The use of ports to make connected gadgets is an adaptation of the technique described by Noble and Runciman in \cite{11}, although our use of ports is slightly different. In our implementation, the functions \texttt{newport} and \texttt{runApplication} cause side-effects which has consequences for referential transparency. Although it is possible to use
monads to get round this problem, the result would be that the whole construction of a GUI would have to be done within a monad and would then appear to be more of a sequence of commands than a simple functional construction. For this reason we have adopted the ML style using side-effects in a carefully controlled and limited way.

5. Conclusions and Further Work

We have presented a declarative GUI toolkit written in ML for use with the Windows 95 and Windows NT operating systems.

The toolkit and the implementation supporting it present the GUI programmer with a much simpler way of constructing GUI programs by taking care of many of the low level details in interfacing to the Win32 Windows application programming interface (API). We claim that by wrapping up side-effecting procedures in a controlled way to present components as values has benefits for reusability, compositionality and comprehensibility. The details of the use of side-effects in construction and execution of a GUI application program is effectively hidden from the ML programmer. There are several areas where more experimentation is needed to determine limitations of the model and possible extensions. For example, message sending is asynchronous although some synchronisation is possible with modal dialogues. It may well be the case that more sophisticated control over synchronisation may be needed in larger applications, but it is not clear in what form and how it should integrate with the non-blocking message passing and blocking dialogues used in the model. Other improvements to the toolkit involve making better use of the ML type system to enforce the protocol constraints for the Win32 API.

We believe that functional programming languages like ML offer a lot of advantages in teaching the implementation and prototyping of user interfaces and we intend to use the toolkit in University classes at Brunel University to evaluate its teaching value.

Bibliography


[5] Brad A.Myers, *Why are Human-Computer Interfaces Difficult to design and Implement?*, Computer Science Department, Carnegie Mellon University, July 1993


Acknowledgements

This work was support by an EPSRC TCS grant in conjunction with Abstract Hardware Ltd (http://www.ahl.co.uk). Many thanks to Simon Finn for his technical support and contribution to the Windows version of the runtime system of ML, Nic Chapman, James Margetson and Colin Saunders for the ideas they contributed to the project, Mick Francis for his contribution to the assembly module of the runtime system, Chris Read for sharing his knowledge on X-windows and windowing systems and Roger Hughes for the useful discussions we have had.