Graphical User Interfaces for Haskell

Duncan C. Sinclair[*]
University of Glasgow

Abstract

User interfaces are normally based on low-level trickery either within the run-time system, or in a separate program which has been connected to the stream I/O system of the language. We present a new twist to this by giving some intelligence to the outside system, which will have greater control of the interface. This has a number of benefits: it makes creating new programs easier, increases the efficiency of the resulting system, and improves the separation between the two halves of the system.

1 Introduction

Many people have written of the problems functional languages have with user interfaces, and have proposed various solutions. These solutions range from the simplistic [9], to the powerful [10], with some truly innovative possibilities explored [4,5].

These systems usually have, at some level, the functional program communicating with an external system, receiving events from the user or system, and replying with requests, telling the system what to do next. This can be done either as low-level run-time calls embedded within the language, or as an external process, connected to the input and output streams of the language. This difference is of no concern, what matters is that at some level there is a protocol between the program written in the functional language and another system which acts as its agent, creating and manipulating the interface.

Mostly, however, control is held firmly by the functional program. We wish to pursue the idea of the user interface being controlled by an external agent which has its own intelligence, and can be programmed separately from the functional program.

2 Tcl and Tk

Tcl and Tk provide a simple yet powerful programming system for developing windowing applications. We will use this system for our externally-controlled interface.

2.1 The Tcl Language

John Ousterhout's Tcl [6], which stands for "Tool command language", is a simple interpreted language, intended to be extended and embedded within an application. Its purpose is to provide a means by which systems may be controlled by users and programmed by the application writer.

[*]Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK.
E-mail: `sinclair@dcs.gla.ac.uk`

Tcl has strings as its only base type. It can arrange these into arrays or, if they are numeric, regard them as numbers. Here is a small example program to calculate the factorial of 10:

```
proc fac x {
  if $x==1 {return 1}
  return [expr {$x * [fac [expr $x-1]] }]
}
set a 10
puts stdout "The factorial of $a is [fac $a]"
```

Square brackets cause in-line evaluation. Curly brackets are a form of quoting, usually used to hold program fragments which will be interpreted in a recursive manner.

2.2 The Tk Toolkit

Tk [7], also by John Ousterhout, is a toolkit for the X Window System [8], based around the Tcl language. It allows the creation of user interfaces built out of components such as buttons, menus, and dialogs. This can either be done in a imperative language such as C, or in Tcl. One feature of this is that it is possible to write complete programs in Tcl, using Tk for its interface. It is also possible for users or external processes to control Tcl/Tk applications using the Tcl language.

Here is a very trivial Tk program, written in Tcl:

```
label .hello -text "Hello, World!"
pack append . .hello {}
```

Without going into too much detail, this creates a small label which says “Hello, World!”, and displays it in a window.

This two-line script is at least an order of magnitude shorter than the equivalent in C, or any functional system where the interface to the window system is at a similar level to that of C. This makes writing user interfaces much easier than before, and with Tcl at hand no power is lost.

2.3 Multiprocessing

The programmer’s first introduction to the Tcl/Tk system is through a simple shell, called `wish`. It can either be run interactively, for experimentation and debugging, or in a batch mode, submitting scripts to the interpreter. Normally, the script will set up an interface, with some action procedures to be executed when buttons or menus are activated. Once the script has been evaluated the Tcl interpreter, rather than exiting, waits for any other commands to arrive from the user or other external processes. This multiple input scheme can be thought of as some sort of multiprocessing. Such pseudo-multiprocessing can be thwarted if any “process” gives the interpreter a command which does not terminate.

2.4 Extending the Language

By modifying the `wish` shell with a little bit of C programming, we have created a new shell program which have called `swish`, with extra commands added to the Tcl language.

We have added three commands, the first, `spawnchannels`, is responsible for spawning an external process, creating three communications channels (i.e. pipes) between the existing and new processes.

The first channel feeds straight into the Tcl interpreter, allowing the external process to feed commands to the running Tcl/Tk program. This can be anything from supplying a complete Tcl program for execution, to the occasional procedure call to update state or modify the program's appearance.

The other two commands are tied to the remaining two channels, and allow messages to be sent back from the Tcl program to the external system. We partition these messages into asynchronous events (the `event` command), and synchronous replies (the `reply` command). These are sent on independent channels to help avoid deadlock, which would otherwise have to be solved by some process of selecting, separating, and buffering these messages in the external system (this is especially difficult when the external system is a functional program). One major advantage of this separation is that we can run the two systems in an asynchronous manner, which makes possible true concurrent operation.

3 Haskell with Tcl

We will now look at what happens when the external process is actually a Haskell [1] program. We chose Haskell because of its good I/O primitives, and good support from the language designers and implementors.

3.1 Process Communication

To `swish`, the Haskell program is the external process, but naturally it works the other way around from the point of view of the functional programmer. So in this section we will talk of the Tcl process as being the external process.

Using the Haskell optional request ReadChannels, and the standard AppenChan request, a Haskell program can communicate with our external process through the three channels created by the `swish` program. The Chalmers Haskell B compiler [3] also provides 'TICK' and 'TIMEOUT' channels which are useful for creating "real-time" graphical programs. Figure 1 show how this works.

We have not yet investigated the best way to structure the functional code, and admittedly early efforts have been difficult to program and read. It is for this reason that we omit a sample of what such a program looks like.

The structure of the Haskell program we use is similar to the typical event-loop found in imperative languages, but unfortunately can get complicated because of the amount of state and channels being passed between functions. Using mechanisms similar to that in the Concurrent Clean system [2], it may be possible to structure the functional program in a cleaner way.

3.2 Examples

Typically, the Haskell program takes no part in the layout of the interface, and how it works; all this is left to the Tcl program. The Haskell program deals with higher-level decisions, such as what information will appear in various windows, while the Tcl program is left to decide how this is done.

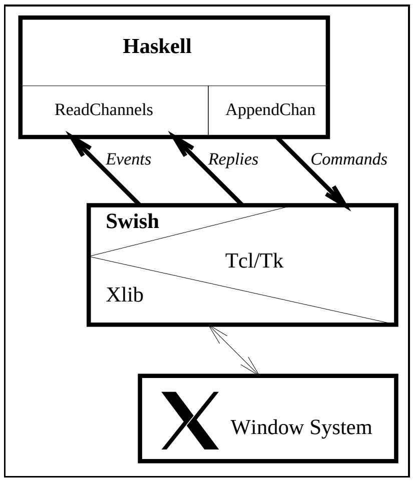


Figure 1: Communication between Haskell and Tcl/Tk

An Alarm Clock

For our first example, we have a simple alarm clock program, in which the Haskell program keeps note of what the time is, and when it should activate the alarm. Every second it advises tcl what the time is, using a procedure defined in the script that `swish` has executed. The Tcl/Tk process then updates the display, without the Haskell program knowing whether it is running an analogue or digital clock. When the user sets the alarm, the dialog is conducted exclusively within the Tcl/Tk process. When this is concluded, the Haskell program receives an 'alarm set' event, telling it when to activate the alarm.

This shows how a greater degree of separation between the interface and functionality can be reached using this method, compared with other methods where the distinction tends to be blurred.

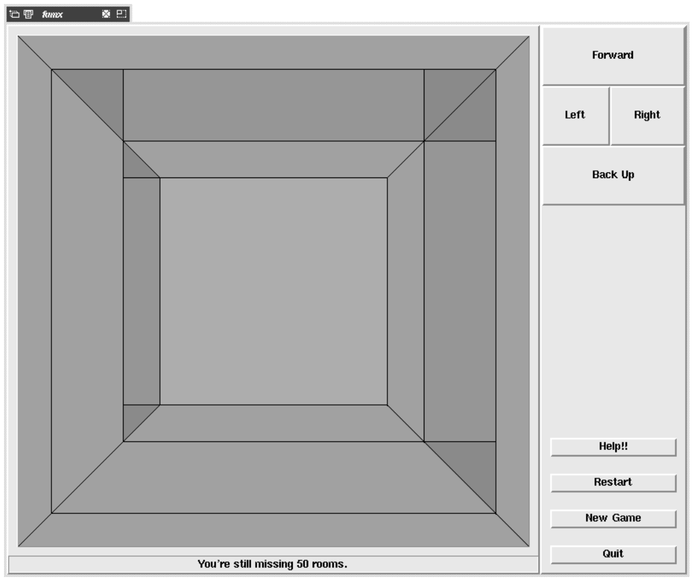
When the alarm is activated, our Haskell program sends a command to the Tcl/Tk process to display a flashing window. This window is then completely managed by the Tcl/Tk process, flashing it every second until the user acknowledges it. Meanwhile, the Haskell process continues counting time.

A Maze Game

As a more substantial example we created a three dimensional maze game, written in Haskell, using Tcl/Tk for its interface. The general idea is for the player to completely navigate the maze, using simple commands such as turn left, turn right and move forward. An indication of the separation between the interface and the program is that the two halves were written by different people in different countries.

The Haskell program is responsible for looking after the creation of the maze, keeping track of where the player is in the maze, and the current view of the maze. It takes events such as 'left', 'right', and 'forward' and causes the display to be updated by sending to the Tcl/Tk process a list of where there are walls visible.

The Tcl/Tk program sets up the display, which includes buttons that the player uses to navigate the maze, plus a perspective view of the maze as 'seen' in the direction the



You're still missing 50 rooms.

Figure 2: Functional Maze in X

player is facing. When a button is pressed by the player, the program passes on the appropriate event to the Haskell process. It also receives the list of visible walls, and updates the display accordingly.

Neither process 'knows' what the other does with messages sent, and either could be implemented totally differently, without affecting the other.

Figure 2 shows what the maze program looks like. A copy of the source may be requested from the author using electronic mail.

4 Discussion

In the abstract we claimed that our system would do three things for us: make programming the system easier, make it more efficient, and improve the modular separation between the functional program and its interface.

We claim that the Tcl/Tk system is easier to program than typical other systems for creating interfaces. One major reason for this is that the interface is written in an imperative language, which we would argue is more suited to interface creation and manipulation than functional languages. Our evidence for this is simply that we find it hard to create user interfaces within our functional programs. In our system almost all of the interface handling is transparent to the functional program.

We feel it is more efficient, simply because two heads are better than one. Even in other two-process systems what usually happens is that one process is always waiting for the other. They run synchronously, to ensure that they can predict what the next input will be. Also, the system external to the functional program is usually not smart enough to do anything useful. With our Tcl/Tk system, we can program it to do other things when not doing anything else. The flashing alarm is an example of this.

In any system it is good for the user interface to be separate from the main functionality of the program. This increases portability between systems where only the interface would need rewritten, and helps keep interface decisions out of the main body of code where it may lead to problems later. Our system clearly helps in achieving this aim by having the user interface defined in a totally different language than the main body of code. We feel that this demarcation is a good thing, with functional languages being used for the parts of the programming task to which their special abilities are suited.

Acknowledgements

Thanks to: John K. Ousterhout, the author of Tcl and Tk; Satnam Singh, for motivating me to do this; Gilbert Cockton, for motivating my desire to get it right; Patrick Sansom, who helped on a previous version of the system; and lastly Carsten Kehler Holst who motivated and implemented the Haskell part of the maze example.

Bibliography

1. P Hudak et al, “Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27 (May 1992).
2. P. M. Achten, J. H. G. van Groningen & M. J. Plasmeijer, “High level specification of I/O in functional languages," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland, 1992.
3. Lennart Augustsson, “Haskell B. user’s manual," From Haskell B distribution, Aug 1992.
4. Magnus Carlsson, “Fudgets – A Graphical Interface in a Lazy Functional Language," Draft, Chalmers University, Sweden, August 1992.
5. Andrew Dwelly, “Graphical user interfaces and dialogue combinators," ECRC, 1989.
6. John K. Ousterhout, “Tcl: An Embeddable Command Language," in *Proc. USENIX Winter Conference 1990*.
7. John K. Ousterhout, “An X11 Toolkit Based on the Tcl Language," in *Proc. USENIX Winter Conference 1991.*
8. Robert W. Scheifler & Jim Gettys, “The X Window System," *ACM Transactions on Graphics vol. 5, No. 2* (Apr 1986).
9. Duncan C. Sinclair, “Graphical User Interfaces from Functional Languages," Final Year Project, May 1989.
10. Satnam Singh, “Using XView / X11 from Miranda," in *Functional Programming, Glasgow 1991*, Workshops in Computing, Springer-Verlag, Aug 1991.