UNIVERSITY
*of*
GLASGOW

# Computing Science

*Master's Thesis*

# Interacting with Functional Languages

*Duncan Cameron Sinclair*

Submitted for the degree of

Master of Science

**Interacting with Functional Languages**
by
Duncan Cameron Sinclair

Submitted to the Department of Computing Science
on Thursday, 1st May 1997
for the degree of
Master of Science

**Abstract**

Functional languages are mathematically pure, and easier to reason about than their imperative rivals. Because of this, they are an attractive paradigm of programming. They allow programmers to express complex algorithms in a declarative manner, resulting in powerful programs that are also well written.

Good interaction with users is important for programs which are to be used for real applications. This normally involves creating a user interface using devices such as menus, buttons and scrollbars. It is now increasingly common for the interface of programs to be developed using specialised tools allowing a faster development cycle, with less programming involved.

In the past, pure functional languages have been poor at creating graphical user interfaces resulting in good applications with poor interfaces. This is due to the mixing of the user's world which involves complex multi-level interactions, with the functional world which has a single threaded state. This is not a very good abstraction of the world to interact with. When the traits of the user's world are introduced into the functional world it is found that the purity of the functional world is compromised, and the clean declarative style of functional programming is lost. If the user interaction is separated from the functional program, allowing users to communicate with functional programs using external interface programs, it is possible to preserve the natural simplicity of the functional world. This would also allow programmers to take advantage of user interface development tools.

I look at current solutions for performing input and output from functional languages, with particular reference to the Monadic I/O style which is currently gaining popularity. I then present a scheme where I have deliberately separated interaction from functionality, allowing functional programmers to build programs that interact with the "Real World", with less interference of the real world into the pure environment within the functional program.

# Contents

# Preface

### Reader's Notes

This thesis can be seen as having two audiences — one in functional programming research, the other in HCI research. My own background is in functional programming, therefore much of the writing is addressed towards other functional programmers. In particular, the sections on HCI research have been written for an FP audience, and so may appear over-simplified to HCI researchers. On the other hand, I have tried to address the sections on functional programming to both functional programmers and HCI researchers wishing to know more about functional programming.

Source code given in the appendices is available on selected FTP archive sites, or by e-mailing the author, at `Duncan Sinclair <sinclair@dis.strath.ac.uk>`.

### Contribution

The basic contribution of this thesis is to show that using separate interfaces to functional programs is a good solution, and that as user interface development tools improve, it will become increasingly difficult for functional solutions to user interaction to meet users' expectations. I further claim that most external interaction which is not functional in nature is better done outside of the functional environment.

My key goal was to try to meet a list of requirements generated by examination of a number of areas of study outside functional programming while creating a system to allow graphical user interfaces to be created for functional programs. In particular, I wanted to enable the programs created using my system to have a high level of modularity, with relative ease of design and programming. I also regard the extensibility of the resulting system to be very important.

## Acknowledgements

This thesis is the result of many years of unrelated research which went no-where, but eventually brought me back to the subject I worked on for my undergraduate project. The route was tortuous, but a true adventure, and I have few regrets.

Many people have helped me on my journey, in a official capacity; Simon Peyton Jones, Stephen Todd, John O'Donnell, and most recently Gilbert Cockton. Others have helped merely by being there, listening, reading; Aran Lunzer, Carron Kirkwood, Alex Ferguson, Patrick Sansom, Graham Hutton and Deryck Brown. Carsten Kehler Holst, Andy Gill and Simon Marlow contributed towards some of the examples presented in this thesis. The Tcl/Tk system was created by Dr John Ousterhout. I give my thanks to each of them, and to the others, who know why.

A final mention for my parents, Anne and Alan Sinclair, for all they have done for me. I hope their efforts will be repaid with interest.

# Chapter 1

# Introduction

Computing has come a long way since the time when most programs were written in languages such as COBOL or FORTRAN, submitted on paper tapes or cards and run in batches — the output being collected from the computer centre the next day. Now we all have our own personal computer on our desk, and we expect it to 'interact' with us. At the same time, programming languages have developed and there are now many paradigms of programming languages. One such is the functional paradigm, which is based upon mathematical foundations, and allows a greater expressive power than previous paradigms.

Unfortunately, the interactive power of these functional languages has found it difficult to keep up with the interactive systems in common use. They tend not to offer input and output features even as advanced as the dinosaur languages mentioned above.

The functional programming community now needs to find a way to enable functional languages to be used to build interactive programs that can communicate with users in a way with which they are comfortable. Furthermore, the programmer's task of creating these interactive programs must not be any greater than in any other language — otherwise the programmer may choose to stay with non-functional languages, and lose all the advantages of the functional style.

The introduction begins by examining functional languages, aspects of good interactive systems, and the requirements that will be used to build a better system to provide good interactive interfaces. The chapter concludes with a statement of goals and a sketched outline of the remainder of the thesis.

## 1.1 Functional Languages

Functional languages are mathematically pure. This means that programs written in a functional language can be manipulated and reasoned about in the same way as mathematical functions. The particular property of functional languages which allows such equational reasoning is called referential transparency, and guarantees that any particular function when called with identical arguments will always return the same value. Although this may not appear immediately useful to all programmers, it is extremely useful when they come to compile their programs — the compiler can perform mathematical transformations knowing that it will not change the meaning of the program. A more obvious direct benefit for programmers is that if they wish to show that particular portions of their code are correct according to some higher-level specification, they can use equational reasoning to prove properties of their program.

Functional languages are declarative. This means that rather than specifying a program as a sequence of operations to be performed, as is usual in 'imperative' programming, the program is written as a description of the result desired. This can be illustrated with a simple example. In a non-declarative, imperative language, in order to sum a list of numbers it is necessary to explicitly keep a running total, accumulating a sum as each element of the list is examined. In the declarative style, the program would express the sum of a list in a mathematical way which might imply the same operations as for the imperative code, but without the programmer having to think at that lower level. Put succinctly, declarative programming means you program 'what', rather than 'how'.

There are a number of particular features that tend to be found in all modern functional languages. The most obvious is the Hindley-Milner type system [19] which provides a powerful, flexible type system, with sum (similar to variant records in Pascal) and product (tuple) types, and flexible polymorphism. It also makes explicit typing optional, as normally the types of all functions can be inferred from the context by automated analysis. This rigorous type system is extremely powerful and flexible, and will guarantee that all programs that are type-checked cannot fail due to a run-time type mis-matches, a problem that is too commonly found in programs written using languages with less rigorous type systems.

A good case for the importance of functional languages was made by Hughes [17]. In

his paper, Hughes puts forward two compelling reasons why functional languages are interesting and useful. Firstly, he shows how functional programs have compositional properties based on simple function composition that allow large scale code re-use by allowing major sub-programs to be bolted together in a flexible manner. Secondly, he points out that, by writing small functions which are not specific to a particular data type, i.e. they are polymorphic, these functions can be re-used with lots of different types. Simple examples of this include list map, fold, and filter, all of which take a function and list as arguments, and will apply the function to elements of the list in different styles.

Functional languages by their nature are particularly good in applications which involve some process of transformation of input data into output data [30]. Compilers are an obvious and well exercised example of this [3,12].

### 1.1.1 Functional Languages and User Interfaces

This thesis is about providing user interfaces for programs written in functional languages, and whether functional languages make this easy. Unfortunately functional languages do not seem especially well suited to the task. Currently creating good user interfaces for functional programs is a difficult task, harder than it is for unsophisticated imperative languages like C. I intend to find out why, and to put forward a system which will allow good user interfaces to be used for functional programs.

Why do I want to do this? With all the features and benefits made available by choosing to program in a functional language, it would also be advantageous to be able to interact with programs written in functional languages in the ways programmers are used to. Compromising the interface for the sake of being able to write in a nicer language is not a viable option for real applications.

Writing traditional interactive functional programs has always been a problem; in 1985 William Stoye gave a brief discussion of this [34]. Over the years there have been two main contenders for how to do input and output from functional languages which allow interaction, first the 'streams' or 'dialogues' approach outlined by Stoye. In this system the result of the functional program, instead of being a conventional type such as a number or a list of characters, is a list of commands which would perform various input or output operations. Results of these actions, including user input, are supplied in a list

as an argument to the function.

The second approach is that of 'continuations.' This works by providing functions which will carry out input and output requests and then ensuring that these functions are evaluated in a particular order. The order is defined by setting down that the result of each continuation function is supplied as an input to a subsequent continuation. This ensures a linear execution of I/O operations, which keeps the program free of side-effects that can affect referential transparency.

Both continuations and dialogues create a definite sequence of interaction so as to ensure that referential transparency is not compromised. If I/O actions were to be allowed to be performed in an undefined order, then functions could give varying results depending on what I/O actions had preceded their evaluation.

Hudak and Sundaresh [16] provide some good arguments in favour of continuations over the dialogues approach, but still manage to make simple interaction appear awkward. Examples of simple interaction programmed in these two styles are presented in full in Chapter 2. However, as soon as they are applied to graphical interaction, both styles of programmed interaction just break down. I consider this point also in Chapter 2.

Laziness increases the chances that you will not know if one particular section of program will be evaluated before another. This will lead to problems in programming interaction. There are some functional languages, e.g. Scheme and SML, for which interaction is not a problem. This is because they use side effects within a strict evaluation framework. This makes most of the difficulties go away, but at the expense of referential transparency.

Lisp must also be mentioned here. There are many good toolkits and user interface development environments built round dialects of Lisp. However, Lisp is only marginally a functional language and in this particular area it is not very pure in its functions. It does make a good model for what is possible in a non-procedural language.

So I shall restrict my attention mostly to pure, non-strict functional languages where interaction remains a problem. The most obvious instance of these is Haskell[15] with some reference to a similar, but older language, Lazy ML[3]. I shall not investigate languages which have had their type system extended, a potentially expensive option, which buys little over what can be achieved with some clever programming as described in Section 2.4. I shall survey the I/O techniques with which Haskell is supposed to be

able to achieve user interaction in Chapter 2.

Within the context of Haskell some new solutions to the interaction problem have appeared in recent years, notably the monadic approach to I/O [27], a refinement of continuations, and the Fudgets functional interactive toolkit [5], a system based around an enhanced dialogues scheme. Unfortunately both suffer from the basic dilemma which functional programmers face when programming interaction — the functional language has to sequentialise all I/O, while interaction is not naturally sequentialised in the user's mind. As lazy languages are not especially sequential in their execution, it is no wonder that programming in sequentiality is difficult.

### 1.1.2 What Functional Programming is Missing

Why should I take notice of work outside functional programming on user interaction? Early indications show that functional languages on their own aren't doing so well. So much of the current work on user interaction for functional programs is coming from a functional programming point of view. It is typically mathematical, with a high theoretical content and little regard to research in other areas.

While it is fine that monadic I/O has good theoretical under-pinnings, did anyone stop to think if it helped in user interface programming? If all that it is is a means *somehow* to construct interactive programs, then it succeeds, but that is not enough. A programmer chooses a language based not on only on ability, but also usability.

When the Fudgets system was created, some HCI user interface research was consulted and as a result good interfaces can be constructed using the toolkit. Unfortunately some software engineering and HCI principles were missed, leading to a poor programming style which lacked the flexibility required to allow easy iterative design of interfaces.

Clearly these new approaches have not delivered large interactive programs yet, and as shall be seen in Chapter 2, there is some doubt that they will. Therefore I shall present an alternative approach that draws some basic concepts from software engineering, human computer interaction, and user interface design.

## 1.2 Approaching Interactive Systems

When creating interactive systems there are three areas of computing science research from which programmers might learn something: Software Engineering, on how programs should be constructed; Human-Computer Interaction (HCI): on how an interface should be adapted to the user; and User Interface Software Technology (UIST), on how an interface should be programmed.

### 1.2.1 Software Engineering

I shall address two prominent aspects of software engineering. The first is that it is difficult to write large and complex programs. The second, by far the more important, is that the programs, once written, have to be maintained. As functional languages mature, and programs written using them grow older, the ability to adapt functional programs to work with the latest technology or requirements will be very important.

In civil engineering terms, in order to maintain any newly built bridges, it is important that the correct construction techniques have been used, or else they will fall down. It is no different when a programmer sets out to build a new program.

#### Modularity

The number one principle of software engineering is modularity, which involves the concepts of coupling and cohesion.

Coupling is how much any particular module depends upon the implementation or services of another. By reducing coupling to a minimum, it is possible to change particular implementation techniques within a program without affecting the behaviour of other parts that need not be concerned with such detail.

Cohesion relates to how specific any particular code section is to one particular task. When maintenance time comes around, it is easier to modify a section of code which does only one thing, rather than a number of related or even unrelated things.

Separation has always been an important principle in HCI, which advocates reducing coupling between the user interface and the 'functional core' of a program — that is, the part of the program which takes no part in communicating with the user, but which is

responsible for all of its functionality. To avoid confusion with functional languages, I shall term this the 'application core' or, more simply, the 'application.'

Whether this separation is possible in practice is a matter of debate but as a general principle of design it is difficult to argue with. By separating the interface from the application a number of benefits are immediately to be found. Portability and isolation of change are two principal examples. Further discussion on separation follows below in the discussion on UIST issues.

HCI has learnt from software engineering, and the functional programmer wishing to write user interfaces must do so also.

#### Extensibility

As there are new and better ways of doing things always being found, there is a need to program in a certain amount of extensibility into programs, to allow them to grow and keep up with the rest of the world. Therefore system designers should not limit the scope of their programming systems, but instead build in a certain amount of flexibility which can be taken up later.

To achieve a good level of extensibility there are no simple concepts such as cohesion as a guide. It is more a principle to keep in mind during the design of any system, that features may be required to be added to the original design at a later date.

Extensibility is also very important in the design of a language. The original designers of the Haskell language did not allow for programmer extension to the I/O system, and so many of the things that were required of the language were just not possible. This was one of the motivations for the monadic I/O system that I discuss later on.

#### Portability

It is sometimes necessary that programs have to be moved to new operating systems and computers. It is desirable that this can be done with as little re-programming as possible — at least, no more than for similar code written in an imperative language. Also, as current fashions change, the style of user interface required will change — functional programming needs to keep up with this.

In the discussion of modularity above, I gave one example of how portability can be

increased — by separating system-dependent (user interface) code away from system-independent (application) code.

In designing the I/O system of a language, it is important that the designers choose a set of primitives which can be supported on different architectures but which, at the same time, can exploit particular features of whatever architecture is in use at the time. This brings back the discussion on extensibility — that designers must plan for features which are not currently available on one particular architecture.

So portability is again all about an attitude of programming. It is something to be borne in mind during the design and implementation of any program.

**Summary**

The programs written today may be in use many years from now. If they cannot be maintained, then when change comes, new programs will need to be written. As this is an expensive job, it is essential that programs are easy to build upon.

### 1.2.2 Human Computer Interaction

HCI research is still an inexact science. There are major limits to current knowledge of how humans interact with computers and how this knowledge should affect the design of user interfaces.

Study has shown that in order to cope with this problem, development of user interfaces needs to be experimental, with prototypes being constructed and tested, leading to a iterative design strategy.

There are three important issues here, for design must be:

- Iterative.

  Construction of user interfaces is normally an iterative process. An initial interface is created and then evaluated by its designer. Based on the testing it can be enhanced or changed as required until it is satisfactory. This can be a time-consuming process. If the interface is written in a compiled language, it has to be re-compiled on each iteration. If a small portion of the interface is under test, quitting and restarting, and then re-navigating the interface to get to the area under test will take time if

this is what is required. Once its designer is happy, more expensive user evaluation would be required which will cause yet more iteration.

- Participative.

  As the design of a user interface develops, it is important that the eventual users of the system, or representatives of them, are able to try out the system and make suggestions for possible improvement. During this time the aims of a particular interface may be changed dramatically as user-feedback could result in a complete re-design, or re-engineering of a design. Limitations in a programmer's toolkit cannot impede this. For example, it will not be a sufficient reason for why a button cannot be moved from where it currently is situated, why it cannot be part of a menu, or why it cannot be based on some other style of interaction.

- Exploratory.

  As an interface develops, during iterative user evaluation, it becomes apparent that the space of possible interfaces to a particular program is huge, and that there is always more than one way to meet required features. Thus this space of possibilities must be attacked in a exploratory manner, with the ability to retract and try other avenues of design. Again, this needs a certain inherent flexibility in the programming system used to create the interface.

Generally this is simply requiring some of the same things I stated as good software engineering — modularity and extensibility. Because of the participative aspect, it is difficult to formalise the design of interfaces, as users tend not to think along logical lines of specification and refinement.

### 1.2.3 User Interface Software Technology

By taking current thoughts from HCI research and using these to create tools to help build user interfaces, this is UIST. UIST may be thought of as an applied branch of HCI; where HCI concerns how it should be, UIST reveals how it really is for current technology.

UIST is built upon established HCI principles. From UIST a number of assumptions about computer users can be made.

- They are opportunistic.

Users have a habit of trying to do more than one thing at a time. When given a system flexible enough to allow multiple tasks to be tackled at the same time, users will often advance more than once task concurrently.

- Users vary in the way they like to do things.

  There is a great variation in the preferred means to achieving any particular aim. Programs should not constrain users to doing a particular task in one way when there are other ways possible. A simple example is the order buttons are pressed or text fields are filled in in a dialog box. Therefore my system should be 'user-driven,' where tasks are advanced under the control of the user.

- Users make mistakes.

  This is obvious. Users must be allowed to go back and correct mistakes, especially if they are critical and subsequently would be irreversible.

- Multiple views of objects aid user understanding.

  Allowing users to examine data in more than one way, perhaps even allowing complete display rearrangement, will help them control and understand the information they are manipulating. An obvious example of this is the Macintosh Finder, where icon positions may be moved around to taste, or different styles of listings of files are allowed.

- Users like to be in charge.

  The user should be in charge — the interface should act for them rather than for the program. If the interface seems hostile or sluggish as a result of the program taking control away from the interface, then this will result in user frustration. The interface is allowed to take charge if it needs to ask pertinent questions at appropriate moments. Interfaces must also allow some amount of tailoring for the user — this must be immediate, allowing experimentation by the user on a completed program.

Overall, these aspects of interface design make one thing clear — graphical user interfaces cannot be programmed in the same way as batch or older interactive paradigms, such as command line interfaces or menus. The most important aspect is that graphical interfaces require non-linear control; there is not a single thread of control that runs

through the program, but instead control moves around very quickly, in short threads of interaction which intertwine.

### Separation

An important theme which has already been touched on and will appear frequently through this thesis is separation. This is quite a simple concept — that systems doing different tasks should be kept apart. In UIST this means that the user interface component of a program should be a separate part of the application and not intermixed with other functional parts of the program. In software engineering separation is expressed as a lack of coupling between modules, as discussed in Section 1.2.1. The need for separation is made in many other places in this introduction, but more general reasons involve the limitation of user influence within the core application, allowing easier data checking, and the ability to alter the interface without reference to the core functionality.

The UIST literature has much in the way of discussion about separation. Edmonds' survey[9] is to be noted in particular, with the collection edited by Pfaff[28] essential reading for UIST researchers. Cockton's thesis[7] has a whole chapter addressing many aspects of separation in the context of user interface management, which contain many useful references.

One point to note is the degree of separation. Whether it is possible to achieve complete separation, where different user interfaces may sit on top of the same back end without changes, is a current area of research. For the purposes of this work, it is accepted that in the systems presented the separation will not be complete in this sense.

The dilemma is that, for the user interface to be completely separate, it must be totally ignorant about the particular style of implementation for a particular functionality, but at the same time this can be essential knowledge for the interface to be able to communicate with the application. This also works the other way around. The application should not be aware of the mode of interaction that the interface presents to the user. However some aspects of the functionality may be required to be structured in a particular compatible style.

The solution is that either the separation is compromised, or else a third agent is required which is allowed to be knowledgeable of the interface and application, while

the latter two are ignorant of each other.

Some further exploration of this area can be found in the conclusions to this work.

### Non-linearity

In order to program the non-linear interaction model which graphical interfaces produce, new styles of programming have been developed.

- The Main Event Loop

  In this scheme the complete program is controlled by a single loop which dispatches control to various sub-routines based upon user events such as button clicks or key presses.

  The major benefit of this approach over localised handling of events is the ability to support the opportunistic approach outlined above. If there is specialised event-handling code for each part of the interface then, as the user switched their attention around, control would be transferred between the various event-handlers. This, however, would be difficult in practice and, without extra code, would be required to handle the switches in context and ensure consistency.

  Having the event control in one place enforces a certain amount of consistency in event handling, ensuring that controls work in the same way between different parts of the program. The down-side is the programming bottleneck that the main loop becomes. As new user interface components are added to a program, the handling of them all must be added to the main event loop. This is an issue for team programming and in later maintenance of the code.

- Callbacks

  The callback scheme is an abstraction over the main event loop — the main event loop still exists, but it is not directly programmed by the interface creator. This goes some way to removing some of the problems of the main event loop. A callback is a function associated with a particular event.

  As new user controls are added, callbacks can be registered for particular events happening in that particular area of the screen. Then when event loop receives an event, by determining where on the screen it happened, it can go to a table of

callbacks and pass control back to the interface component which registered the event for handling.

This results in an overall increase in modularity, as the main event loop can be programmed without any knowledge of the structure of the rest of the program, and so there is looser coupling in the final program. The only possible negative aspect of this is the lack of intimate control over the main event loop which may be required in a particular application but usually there are hooks to help cover this.

Both these schemes give a crude form of multi-threaded program execution as different parts of the program are able to do their own independent actions without conflict from other parts of the program.

#### Window Systems

Modern interfaces on conventional computers now tend to rely on high resolution displays, typically sub-divided into "windows", controlled by the user using a standard keyboard, plus a mouse, allowing direct manipulation of the display.

Typical abstractions used with this type of display are buttons which trigger specific actions, or act as flags; scrollbars which allow the display of large windows within smaller windows, allowing the area in view to be changed; pop-up menus which allow grouping of functions; and icons which allow objects to be manipulated by the user directly.

Of course interacting through windows and with a mouse is by no means the only way, with pen and voice based input increasing in popularity but, as it is currently the only common style, I shall concentrate my efforts here. Any user interface system built for functional languages should be sufficiently extensible to allow work to be carried over to new paradigms of interaction.

#### Toolkits

It is normal to build toolkits to harness the raw functionality of windows and graphics primitives to be used to construct interfaces and manage callbacks.

As with many parts of computing, toolkits are concerned with providing abstraction to make the power given more controllable. Typically, a set of "widgets" will be provided

which supply a bundle of functionality in a window on the screen and will employ callbacks to allow the programmer to assign specific functions to them.

Widget sets usually include buttons, menus, scrollbars, and text entry boxes, plus higher-level widgets used to structure on-screen appearance.

**Review of UIST Techniques**

UIST is all about making key programming tasks for user interface creation easier. The most important aspect of this is the user interface separation principle, but there are other specific tasks frequently expected of a UIST tool.

- Multiple Active Threads

  The program should be able to give the impression of being able to do more than one thing at a time, as this is what the user will require.

  As mentioned above, this is normally done by structuring the program into small independent units — widgets, running from call-backs from the main loop of the program, and suspended as other threads run. Naturally, this requires each individual widget to manage its own state between calls.

- Imperative Control Structures

  When there are particular sequences of dialog or interaction, these would be controlled by the UIST framework. Thus standard imperative control structures of looping, branching, conditionals and sub-routines for particular interactions when required, must be available.

  In some cases a tight loop of communication between the interface and application is required to ensure that user inputs conform to program-set requirements. This is called 'semantic feedback', as the interface itself only understands user syntax and requires the application to decide on the validity of the input.

- Multiple Views of Data

  An important feature of many interactive programs is that the program is able to present multiple views of the same data; for example, a text editor might allow multiple windows to be open on the one file, with updates in one window would

be mirrored in the other. Another example is the 'fat-bits' bitmap editing available in various graphics packages, where fine pixel editing is also reflected immediately in the normal size image, perhaps in another window.

Such features tend to be implemented either with a centralised abstraction on the data, which then controls the multiple views from inside, or by a distributed system in which the multiple views are kept in tune with each other by broadcasting events from one view to all other views.

Naturally different toolkits and UIST tools will implement these tasks to different degrees, with simple toolkits doing much less than a full user interface management system but, for complete flexibility and simplicity in small productions, it can be better to use a simple toolkit, rather than committing to a all-encompassing UIST framework.

### 1.2.4 An Example Toolkit — Tcl/Tk

I shall now present briefly one particular toolkit to show how toolkits in general work, and what sort of features they provide. This toolkit — Tcl/Tk [25] — will be used in Chapter 3 to provide interfaces for functional programs.

John Ousterhout's Tcl [23], 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.

Tk [24], also by John Ousterhout, is a toolkit for the X Window System [29], based around the Tcl language. It allows the creation of user interfaces built out of components such as buttons, menus, and dialogs.

#### The Tcl Language

Tcl has a clean and simple syntax. It is designed to be able to be used as a user-centred shell for graphical programs. Tcl has strings as its only base type. It can arrange these into lists and lists of lists, etc. Numeric strings can be regarded 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. Braces are a form of quoting, usually used to build lists — especially lists representing fragments of Tcl code, which can then be interpreted in a recursive manner. In these respects Tcl is very similar to Lisp, with a nicer syntax. It is even possible to program a system of anonymous higher-order functions, using the standard ways available to the user of the language for extending it.

**The Tk Toolkit**

Tk can be programmed either from a compiled language, such as C, or more usually in Tcl, extended with commands for Tk. This makes it possible to write complete programs in Tcl, using Tk for the interface.

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 and another popular toolkit, OSF/Motif (see Appendix A.1). This makes writing user interfaces much easier than before and, with the full functionality of Tcl at hand, no expressive power is lost. As one might expect, there are user interface builders for Tk which allow user interface creation using direct manipulation, in the style of other toolkits.

Tk interface builders, written in Tcl/Tk, have two advantages over other toolkits. Firstly, the Tk system is creating and manipulating the interface by programming the

widgets with code which will allow them to tailor their own behaviour. This code becomes part of the output of the interface builder and can be used at any time to modify the interface created. Secondly, because of the ability of Tcl/Tk to take commands from other input sources, it becomes possible to manipulate the running program, even when it is fully implemented. Therefore, interface experimentation and manipulation can be combined with interface evaluation.

**Summary**

Tcl is a simple clean interpreted language. Tk, designed to be used with Tcl, is a powerful toolkit which is easy to program. I shall be looking at other aspects of Tcl and Tk in Chapter 3.

On the basis of what can be expected from UIST, Tcl/Tk is not a complete solution. However it does provide many of the features and it can be built upon to come closer to a full UIST design.

Add-on tools for Tcl/Tk include extensions to the Tcl language to make it object oriented. There is also a sophisticated user interface design environment which, by using features in Tcl/Tk, can edit the interface of an application it created live, while the program is running. With the power of interpreted Tcl, it is possible to run the interface, while the design program allows simple programming tasks such as menus and pop-up windows to be automated, at the touch of a button.

## 1.3 Implied Requirements

I wish to take the techniques of software engineering, HCI and UIST, and apply these to functional programming. To build a user interface system for functional languages, I will use the implied requirements covered above, and use them to judge the result.

### 1.3.1 Functional Programming Requirements

First of all, it is important that I am able to take advantage of the features of functional languages. In particular, I want my systems to preserve referential transparency. This is perhaps the most important feature of functional languages, and without this, much of the efficiency and elegance of the language would be lost.

On a more pragmatic level, the compositional style of functional programming makes some programming tasks easier. If I were to adopt a style which prevented easy composition of sections of code, then again I would have lost much of the beauty of functional languages.

### 1.3.2 Software Engineering Requirements

Software engineering presents three areas which I need to address.

The most obvious aspect of software engineering concerns modularity. The two basic measures of modularity are cohesion — how much a procedure or function focuses on a single task — which should be maximised, and coupling — how much one section of code depends on another's implementation — which should be minimised. Functions should be written to do exactly one task and sub-dividing this with local functions is appropriate. Requiring code to be written in one section of a program to maintain incidental data used in another part of the program is a good example of poor modularity.

In addition the maintainability of a program can be affected by how easy it is to extend the program to handle areas not originally considered when the program was designed and written. Obviously if any of the tools which have been used in the program's construction are extensible this will help.

Finally, I might wish to move my program to different systems, perhaps to different operating systems, with different capabilities. Problems here can be guarded against by keeping the design of the program as independent as possible from particular system features, and also isolating interaction with external systems into a module which can be rewritten without requiring the rest of the program to be adapted.

### 1.3.3 HCI Requirements

The processes involved in creating good interactive programs tend not to be formal, but more experimental. I must allow interfaces to be built in this way also.

The most important aspect of interface design is that it is iterative, and will not be correct first time. Prototype interfaces must be created, and evaluated, and the evaluation used to create further prototypes. This process can cycle for many iterations.

The design process is iterative, participatory and exploratory. It is essential that

whatever system I devise for user interface creation can support this style of design.

If the interface is tightly coupled to the application code, then adapting it during design would be difficult. Plus, it must be possible to adapt the user interface rapidly, to allow users to try out different styles. An interface which required major programming to be done during each iteration would not permit this kind of exploratory design.

### 1.3.4 UIST Requirements

The requirements suggested by UIST are more demanding but, as such, help me find a working solution to the interaction problem with functional languages. Without these strict requirements, I would not know if I were solving the problem or not.

The UIST framework, being tied up in the programming of the system, has many requirements in common with what is demanded for good software engineering. There are, however, some specific features that are required.

The interface must run as a logically separate component of the program so that it can be developed and maintained separately. There must be, however, a high level of communication between the interface and the rest of the program so that the user gets a realistic notion of what the program is actually doing. If the interface is too loosely coupled to the main program, it becomes difficult to transmit a complete picture of the state of the program to the user, semantic feedback becomes difficult, and the user would lose control over the execution of the program.

I must be able to build up sequences, with loops and jumps in them, to correspond to particular dialogues with the user. This level of dialogue sequencing belongs completely in the interface component of the system, and should not appear at all within the application core.

As most interfaces give the user multiple choices as to which task to pursue at any one moment, while still allowing rapid task switches, it is important that some form of multiple threading of execution takes place. It should be possible for one part of the display to be updating, while the user is still able to interact with another.

Error recovery is very important. The user must be able to cancel a dialogue or recover from inconsistent data with the program state being recoverable. At the same time, undoing critical actions is an important feature if it can be achieved.

### 1.3.5 Overall Requirements

I shall now review some of the notions that have come up in the discussion of requirements, framing them as overall requirements as they are demanded by more than one aspect of my investigation, and so might be regarded as principle requirements.

- Ease of Design

  The framework used to build interfaces must make the design process during interface creation easy. So the toolkit I use needs to be well designed, with the appropriate level of functionality, without being overly complex. Also the connection between the interface and the application requires a high level of flexibility to cope with whatever information the interface design will require.

- Ease of Construction

  It is important that the task of programming the applications for the interfaces designed is made as simple as possible. Without this, people will not be motivated to use functional languages, preferring approaches with which they are familiar. One way to help ensure success is to make interface manipulation as easy and as similar to existing systems as possible with, of course, improvements on this desirable.

- Portability

  I wish to make my interactive programs easy to move between different systems. Interface separation helps in this. Also, the design of the interface system should avoid making assumptions about the target environment.

- Extensibility

  I have emphasised how important it is to build programs so that they may be maintained easily. This requires not only rigorous design and implementation methods, but also that the tools with which I build are able to grow with requirements.

  With respect to interface creation, this might imply a toolkit approach allowing the basic system to be built upon with new widgets, and so allowing the most features for the least programming effort, but the toolkit must be powerful enough to supply enough functionality to create the programs programmers wish to write.

## 1.4 Goals of Work

The main goal of this work is to explore the synthesis possible between functional programming research, and HCI/UIST research. To this end I have been experimenting with the Tcl/Tk system as a tool to create user interfaces for programs written in Haskell.

This work follows on from a series of similar experiments, all of which involved creating separate interfaces for functional programs.

### 1.4.1 Early Work

In the past I have built a number of systems which are meant to help create user interfaces for functional programs. These started at the very basic level of a simple widget which would allow simple drawing controlled by a Lazy ML program. The widget would also return user input back to the functional program. The complete interface was built and controlled from the functional language [32].

The interface ran as a separate process, but there was no intelligence in it. It was separated mainly for convenience, but also to allow the interface to respond to events while the main program was active.

A later system improved on this, using the freely available system Wafe [20]. Wafe uses the language Tcl, as I have here, but gives a different interface style to that used by Tcl/Tk. Its purpose is to allow scripting languages, without the ability to access normal library routines, to create user interfaces.

This system allowed a programmed agent to control the interface, rather than having it controlled only by the application side of the program. This is what I had wanted for my earlier system, and it was the addition of a scripting language to build and manipulate the interface which made the difference.

## 1.5 Thesis Outline

This thesis continues with a study of current I/O solutions available in functional languages in Chapter 2. This includes an examination of a number of proposed systems which try to solve the same problem as I am addressing.

Chapter 3 describes my solution to solving the problem of interaction, with examples of the techniques described.

A full assessment of this work is made in Chapter 4, with a measure of how well I meet the requirements given above.

I follow this with a summary and conclusions in Chapter 5. Source code for the programs discussed in the thesis is given in the Appendices, after the Bibliography.

# Chapter 2

# Functional I/O

This chapter examines the particular styles of I/O in functional languages, with particular reference to the three major styles of I/O used in the language Haskell. These are *Dialogues*, *Continuations* and *Monads*. Dialogues and continuations are standard in the language. Monadic I/O is a extension in the Glasgow Haskell Compiler, but is likely to be adopted as standard in a later version of the language.

Following is an examination into why none of these systems are particularly helpful when it comes to writing interactive programs.

The chapter concludes with a look at some of the current techniques for building graphical interactive interfaces, concentrating on the Fudgets system [5] and Concurrent Clean [1].

## 2.1 I/O Styles

Communication with external systems in functional languages is typically encapsulated within the I/O system so that information can be passed between the internal world of the functional program and the impure world which the external systems inhabit, without referential transparency being compromised.

There are a number of ways of achieving this [16]. Those covered are the two popular systems, dialogues and continuations, plus a newer rival, monadic I/O [27].

### 2.1.1 Equivalence

In his thesis [11], Andrew Gordon shows that these three systems are fundamentally equivalent, in that any one can be implemented using another as a basis without needing any extensions to the language. However, the programming style adopted for each system is very different.

Indeed, in the Glasgow Haskell Compiler system [12], a continuations implementation is built upon a dialogues implementation, which in turn is built upon a monadic I/O system.

### 2.1.2 Haskell Channels

In this thesis, Haskell [15] will be used to demonstrate the I/O systems, as there are implementations of all three I/O systems in Haskell.

Much of interactive I/O in Haskell revolves around the concept of a "channel". A section of the Haskell Report explains this:

> The channel system consists of a collection of *channels*, examples of which include standard input (`stdin`) and standard output (`stdout`) channels. A channel is a one-way communication medium—it either consumes values from the program or produces values for the program. Channels communicate to and from *agents*. Examples of agents include line printers, disk controllers, networks, and human beings. As an example of the latter, the *user* is normally the consumer of standard output and the producer of standard input. [15]

### 2.1.3 Examples used in this Chapter

Two simple examples will be used in this Chapter to help demonstrate the strengths and weaknesses of each I/O system.

The first is a 'minimal interactive program'. The computer queries the user, who will either reply in the affirmative or negative. The computer will then respond with an appropriate reply. Thus there are basically only two possible traces of interaction in this program. Computer output is shown in `this font`, while user input is shown in *this font*.

1. `Do you feel all right?`

   *Yes.*

   `Great!`

2. `Do you feel all right?`

   *No.*

   `Sorry to hear that.`

These example programs will be written in the style of a relatively unsophisticated functional programmer. In each of the programs, further work could be done to improve them, but that would not reflect the sort of program that might be written in real life. An obscure heavily optimised program will reveal little about just how difficult programming in each I/O style can be.

A further example, taken from Cockton's thesis [7], is a simple simulation of the interactions involved with an Automatic Teller Machine. In Figure 2.1 is the slightly modified version of his CSP[1] description, which I will use as the specification of the dialogue.

Programs implementing the ATM dialogue will be of a more sophisticated nature than the first example, reflecting the fact that larger programs require greater effort of programming for elegance. As such, these programs try to represent the particular style in their best light.

It should be possible to implement this example in a staged manner, starting with the event sequencing, introducing the system output, then the user input, without the actual 'application' code of a bank system being present. This order is not fixed, but it appears in practice to have worked well.

## 2.2 Dialogues

A popular system of I/O is that of *dialogues* [21], which is the primary I/O system specified for Haskell. The basic concept is of the program and the system engaging in a sequence of dialogues, the program making requests, and the system responding to them.

[1] CSP (Communicating Sequential Processes) is a formalism for specifying concurrent systems. See Hoare's book of the same name [13].

```
ATM      = InsertCardMessage -> CardIn ->
           EnterPinMessage -> CUSTOMER

CUSTOMER = PinNo -> ValidatePinNo ->
           (  Thief -> KeepCardMessage -> ATM
           [] Wally -> LearnNumberMessage ->
                       EjectCard -> ATM
           [] Retry -> RetryMessage -> CUSTOMER
           [] PinOK -> ServicePrompt -> SERVICES
           )

SERVICES = (  RequestChequeBook -> AcknowledgeChequeBook ->
                                   MORE
           [] RequestBalance    -> ShowBalance ->  MORE
           [] RequestStatement  -> PrintAndProfferStatement ->
                                   TakeStatement -> MORE
           [] RequestCash       -> CASH
           )

CASH     = AmountPrompt -> Amount ->
           (  AmountHopeful -> SorryButMessage -> CASH
           [] AmountOK -> ConfirmPrompt ->
              (  Confirm -> ProfferCard -> TakeCard ->
                            ProfferCash -> TakeCash -> ATM
              [] Cancel -> MORE
              )
           )

MORE     = EjectCard AnotherServiceMessage ->
           (  Continue -> ServicePrompt -> SERVICES
           [] CardOut -> ATM
           )
```

Figure 2.1: CSP specification of an ATM.

The two halves of the sequence of dialogues can be represented within the functional program as two lazy lists, one being constructed by the program and executed by the run-time system of the language, and the other having the opposite property of being created by the run-time system and interpreted by the functional program. The run-time system is the agent responsible for constructing replies to the functional program's request, based upon the response of the user and other systems external to the functional program, such as the operating system.

### 2.2.1 How Dialogues Work

In Haskell, the idea of a pair of lazy lists appears clearly since the type of the main program, a function called `main`, is `Dialogue`, which is defined as:

```
> type Dialogue  =  [Response] -> [Request]
> main :: Dialogue
```

where the result of the function is the list of "Requests" to the run-time system, for which an equal number of "Responses" are generated and returned in a lazy list as the first argument to the program.

`Request` and `Response` are algebraic data types; this basically means that they are tagged union types. The tags are called constructors as they behave like functions which construct values in the type. In Haskell, constructors usually appear named with a capital letter — this is also the case with names of types.

So, the types `Request` and `Response` describe, by their constructors, all the available operations available from the I/O system and the various possible results from these requests. These types are fixed and cannot not be extended by the programmer and so should be flexible enough to meet all possible demands. This is hard to guarantee and it should be noted that in actual implementations of Haskell additional requests have been added to provide previously unforeseen facilities.

Constructions available from the `Request` type include `AppendChan` and `ReadChan`, for appending to the end of or reading from the front of, the data in a channel. The `Response` constructors returned from these requests would be one of `Success`, `Str string` or `Failure ioerror`, success and failure being self-evident and the `Str` construct returning the "string" contents of the channel as a lazy list.

Note that a different approach has been taken for handling input than for output — a difference that can unnecessarily complicate programs. The program can incrementally add to the end of a channel, but can only gain access to the user input as a single list which then has to be maintained in addition to the response and request lists.

### 2.2.2 Example of Dialogues

Figure 2.2 shows a dialogues implementation of the first example. It can be seen that the two lists `input` and `res` have to be carried around between functions, cluttering the

code. Two ancillary functions are used; `die`, which reports an error, and returns an empty list, which terminates the output list and thus the program, and also `lines`, a standard function which is used to split the input list into a list of lines of input.

```
module Main(main) where

main = setup

setup res = ReadChan stdin : getinput res

getinput ((Str input):res)   = how (lines input) res
getinput ((Failure err):res) = die err

how input res = AppendChan stdout "Do you feel alright?\n"
                : okhow input res

okhow input  (Success:res)      = getanswer input res
okhow input ((Failure err):res) = die err

getanswer []        res = []
getanswer (l:input) res =
    case l of
       'y':xs -> good input res
       'Y':xs -> good input res
       _      -> bad  input res

good input res = AppendChan stdout "Great!\n"
                 : okgoodbad input res

bad  input res = AppendChan stdout "Sorry to hear that.\n"
                 : okgoodbad input res

okgoodbad input  (Success:res)      = how input res
okgoodbad input ((Failure err):res) = die err
```

Figure 2.2: Interaction with dialogues.

This program is very hard to read. Replies from requests are handled in different functions from where the request was made, meaning that there is no obvious indication that the two things are at all related. Program flow is not obvious from the program structure and, to make a change to any function within the dialogue, would require changing a number of apparently unrelated functions in order for the dialogue to succeed.

Writing the program was also non-trivial. Too much time was spent on managing the various success/failure results, rather than on the actual dialogue. Various ancillary functions, such as the `die` function, also had to be written.

Counting the number of lines in the complete program, including functions not shown in the figure, leads to the relatively high 39. This is especially high as an equivalent program in C takes 10 fewer lines! It is surprising that the higher expressive power of Haskell does not lead to a more concise program than C.

### 2.2.3 Assessment

The dialogues scheme seems to be universally condemned these days [26,27]. Just trying to manipulate these lists in a straightforward manner, and at the same time getting a well written program working, can be quite a trial.

Other serious problems are:

- Modularity.

  The example shows much constructing and taking apart of lists. It does not show what happens when you wish to call a number of larger functions which wish to communicate through the dialogues scheme. This requires intricate passing around of versions of lists, which can easily lead to typographical mistakes in the code, or space leaks if the programmer is not paying attention. This is an obvious case of increased coupling, as functions depend on each other to handle the dialogues lists correctly.

  In the example, care is taken to separate functions which pattern match on either the input list or the responses list from functions that request output. This is to ensure the output will happen before the input is required. This leads to a great number of small functions in the code, all doing a small part of a larger task. This is poor cohesion, where each function is not addressing one whole task. Extra care would have to be taken if these functions were to be combined to make sure that pattern matching happens lazily or else space leaks could easily occur.

- Flexibility.

  The types of `Response` and `Request` are not extensible, and so programmers wishing to do more either have to modify the compiler, or else implement the required functionality outside of the functional system, and then somehow link it together with the program.

- Effort.

  Every request returns a response — no matter whether it is needed or not. This response has to be checked explicitly for failure. The okhow function in the program is a good example of this, its sole purpose being to check for an error response before continuing interaction. In the next section it will be seen that continuations provide an elegant method for dealing with errors.

  Due to the style in which dialogues programs are written, and the shortcomings listed above, anything more than superficial error handling is difficult.

### 2.2.4 Summary

Even in short programs, the unnecessary complexity of dialogues can be troublesome. For this reason a dialogues version of the second example was not attempted.

## 2.3 Continuations

The *continuation* style of I/O is a similar scheme [16] and can be implemented on top of the dialogue system, as in Haskell, where the explicit request output and response input lists are concealed within higher-order functions. This removes many of the problems listed above for dialogues.

The basic notion behind continuations is that each I/O function takes an extra argument which is the next function to be executed. This is why this I/O system and these functions are normally called continuations — they specify what function will be executed next in an operational sense. This gives a sequential model of I/O which directly parallels the sequential lists in the dialogues model.

### 2.3.1 How Continuations Work

Higher-order functions make the implementation and understanding of continuations more difficult than the simple scheme of lists seen in the dialogues approach, but once understood, continuations can be quite pleasing in their operation.

A continuation is a function which is used as an argument to another function, and is usually 'tail-called' by that function, passing working data. This continuation in turn will

probably have been given at least one continuation which will then be used to continue the path of execution. If a continuation does not take another continuation as an argument, then it must be a terminating continuation. If it takes more than one argument, then it implements a conditional operation, and will choose which continuation to use as a result of the conditional test.

All continuations will have the same result type, which is the type of the main program, and is typically an algebraic or abstract data type. In Haskell, however, it is the type `Dialogue`, the type of the main program, as it was for dialogue-style I/O.

So, the type of a continuation would be something like this.

```
> line ::  String -> StrList -> StrListCont -> Dialogue
```

The function `line` takes two arguments, and then a continuation. The function returns a `Dialogue`.

At this point the reader may be wondering what continuations are doing with the result and responses list embedded in the `Dialogue` type, and made visible by the dialogues style of I/O.[2] This is due to the fact that in Haskell continuations are built on top of dialogues, where each primitive continuation is manipulating the responses and results lists concealed within the type of the continuation.

Internal choice can be given to a continuation by supplying more than one continuation as an argument and allowing it to choose which one to follow. This is a better solution than the programmer having to investigate the result of each operation and build a conditional expression for each action. This solves some of the problems in dealing with errors in the dialogues scheme, in that a continuation may be given a 'success' continuation and a 'failure' continuation, and the appropriate one is followed without further programming.

For each of the requests in the dialogues style, Haskell provides an equivalent continuation. The `Response` type is not seen at all by the continuations programmer, the responses being automatically interpreted by the continuations to return only useful information.

The programmer can then build their own continuations around the primitive operations with their own code encapsulating more primitive functionality, giving a higher

[2]It becomes very obvious at this point that dialogues style I/O is primitive to Haskell, and that continuations are built on top of it. It is a pity that the type `Dialogue` could initially lead to confusion for continuation programmers who should not be thinking in terms of the dialogues I/O style.

level of abstraction to their I/O routines.

### 2.3.2 Example of Continuations

Figure 2.3 shows the example expressed in the continuations style. It can immediately be seen to be more concise and easier to read than the dialogues example in the previous section. This section of code also uses the `lines` function seen in the dialogues example. The continuations `exit` and `done` are both terminating continuations, which return and report, as appropriate, either failure or success.

Slightly more obscure is the `$` function, which can be seen at the far right of the code. This is simply function application, but at a different level of precedence. This allows what would otherwise have to be written as `(f x (g y (h z))` as `f x $ g y $ h z`.

```
module Main(main) where

main = getinput

getinput   = readChan stdin exit                                    $
             \input -> how (lines input)

how input  = appendChan stdout "Do you feel alright?\n" exit        $
             case input of
              (l:input) -> case l of
                             'y':xs -> good input
                             'Y':xs -> good input
                             _      -> bad  input
              _         -> done

good input = appendChan stdout "Great!\n" exit                      $
             how input

bad  input = appendChan stdout "Sorry to hear that.\n" exit         $
             how input
```

Figure 2.3: Interaction using continuations.

In contrast to dialogues, continuations can be very readable. If the `$` function is read as a sequencing operator, which it effectively is, then it is easy to see the flow of control, with the results of operations being handled within the same function as asked for them. The structure of the program is good, split into the simple chunks of interaction.

Writing it was not so easy. Given that it was easier to write than the same program in the dialogues style, it was harder to hold in mind the more abstract concepts involved

in continuations. Working out the types involved in sequencing continuations using the ubiquitous $ operator can be quite difficult without practice. On the other hand modifying this code would be a lot easier than dialogues code. As continuations combine so easily, adding extra actions into a function poses no problem: it is not necessary to worry about affecting the global correctness of the rest of the program.

This program is half the length of the version for dialogues. It is also shorter than the C version. This is more in keeping with what would be expected from the highly declarative style of programming.

### 2.3.3 Assessment

Continuations seem to be easier to use, but can be troublesome to understand as the number and level of higher-order functions tend to grow at an alarming rate. Passing and applying continuations in some cases can easily be forgotten about, leading to type conflicts.

The key to writing continuations-based programs lies in understanding the types. It was found that, when writing a complex continuation which might take a number of continuations as arguments, it is easy to become confused about which arguments a particular continuation needs. Getting this wrong invariably leads to a type error during compilation, often involving functions not immediately associated with the erroneous continuation use, but sometimes in the function which calls it. However, being more explicit with types of continuations tended to both make error reporting from the compiler easier to understand, and actually helped understanding of the use of the continuations.

The following are some of the problems with the continuations style, some of which it shares with the dialogues style.

- Modularity.

  Except for the input stream which, once a handle has been created for it using the `readChan` function, can be thought of as living outside the I/O system, the lazy request and response streams are done away with. They actually still exist, but are handled implicitly by the built-in continuations.

  As the sequential ordering of operations is made more explicit, problems of interleaving input with output is not a problem. Input actions are given as continuations

of the output actions, and so must happen after them.

So, in fact, continuations can produce relatively good modular code.

- Flexibility.

  The continuations style has no improvement over dialogues in this area. Although requests now appear as functions, these functions still map onto the same limited set of requests. In principle, continuations have a high degree of flexibility — they are let down only by their Haskell implementation.

- Effort.

  Here continuations do splendidly over dialogues. By being able to specify a failure continuation as well as a success continuation, the clutter of explicit error checking has been removed. This is the main reason why the continuations style results in a more concise program.

  So, error handling is much improved. Error handling can be moved to a different part of the program where greater attention can be given to it.

### 2.3.4 Formalised Interaction using Continuations

Turning now to the second example of the ATM. This program turned out very clean.[3]

The program was initially constructed by taking each CSP event and turning it into a continuation. System events became continuations producing output, while user events became input continuations, returning a value selected from a data type which corresponds to all the possible user input at that time. The CSP choice operator became a case expression over the data type. User input without choice was naturally handled internally to the appropriate continuation. After the basic structure was in place, the actual (minimal) functionality of each continuation was then coded.

This scheme of programming results in a main section of code which is very similar to the CSP version and has no traces of the primitive functions used to achieve the interaction. In fact this code was written and debugged before the low level input and output code was written. Simple stubs of each continuation were used to test the structure of the program and ensure the types were correct before working on the interaction.

[3] The program listing appears in Appendix B.1.

Next came the output. This was very simple. For each system event a call to appendChan was used to output the appropriate message. Again, this was written and debugged without any input code being written.

Programming the input was more difficult. As already noted, input is inexplicably separated from the I/O system once 'retrieved' using a readChan request returning a list representing the input stream. If this is to be used within a program it must be passed as an argument to, and handled by, each continuation within the program. This lack of inclusion of input lists within the rest of the I/O system can result in code being unnecessarily cluttered.

A different solution was used. Continuations are flexible and easily extended or encapsulated. Normally the result type of the continuations is `Dialogue`. A new `Result` type was created, encapsulating `Dialogue`:

```
> type Result = [String] -> [Response] -> [Request]
```

As well as each continuation getting a hidden 'response' list, it will first get a list of strings. This is the input list split into lines. This can be used and then passed onto the next continuation. Standard system continuations are passed the response list only.

Once this was done and the initial call of the ATM was modified to pass the input list, no other modification of the main program was required. Continuations dealing with input can then use this input list. Output continuations required modifications to deal with the extra argument, but, with the appropriate abstractions, this modification of output code on a global scale could hopefully be kept to a minimum.

These modifications are needed because in a functional language if you want a global state it must be all-pervasive. Rather than being used in only one location, the state must be carried along the path of execution. The lists managed by the continuation functions are an example of a global state and thus when the structure of the global state is modified, all functions manipulating this global state must also be modified.

### 2.3.5 Summary

Continuations, although potentially confusing to begin with, are easier to use in real programs than dialogues. They have a clean nature which results in tidy programs, which are easy to read and debug once the types of the functions are understood. The

ability of continuations to be extended and encapsulated allows programs to be built up in phases without requiring redesign of existing code.

## 2.4 Monads

A new refinement to this system comes with the application of *monads* to I/O [27]. Trivially, the monad system can be seen as a refinement of the continuations mechanism, but with a greater level of encapsulation of control. Monads work with a token which represents the current world and all I/O operations are based on this token. This token is the basis of the monad.

Monads are a categorical concept. Category theory is an advanced branch of mathematics, discussion of which is outside the scope of this thesis. Interested readers are referred to Barr & Wells' introductory text [4]. Further good references concerning monads are the papers by Wadler [37,38].

### 2.4.1 Theory of Monadic Programming

Here is a brief description of what a monad means to the programmer. In the I/O world the general monad used is called the state monad, but in this thesis the term monad will usually be used.[4]

The idea behind the state monad is that it contains all global state that the programmer wishes to manipulate — including the 'world' outside of the functional program. The programmer will then define a number of primitive operations which will create a new world, manipulate it or, more practically, modify it to some desired end. Referential transparency must not be compromised however — for example, by being able to modify part of the global state in one place and having a corresponding effect elsewhere in an unrelated part of the program. In order to preserve referential transparency it is sufficient to ensure that there is only one current state available to the program, i.e. that the state monad is not duplicated, but remains unique through the execution of the program. This implies a single thread of operations on the state monad, each modifying the state it contains in turn.

[4] Another monad the programmer would use, but not immediately recognise as a monad, is the list monad, which defines how lists are constructed, and provides many of the primitive functions on lists.

This aim is achieved by ensuring that all monad operations are linear, i.e. that they use the monad only once (they are passed exactly one monad as an argument) and return it only once (their result type must contain exactly one monad type). Primitives are programmed to do this and other functions cannot break this rule because they do not have access to the actual monad except via the defined primitives.

### 2.4.2 Practical Monadic Programming

At the theoretical level, a monad is defined strictly by two functions, `bind` and `unit`, with types:

```
bind :: M a -> (a -> M b) -> M b
unit :: a -> M a
```

Exactly what these types mean is not especially important to this discussion — `M` is the monad type and `bind` and `unit` are functions which operate of values of this type. These functions are the basis of the principal operations on any monad.

The `bind` function is normally used for sequencing monad operations. It takes the result of one monadic operation then passes it as an argument to the next operation. Often an analogous operation `then` is used. It is strict in its first argument and thus forces the first operation to fully return its result before the second operation is started.

The `unit` function is used as a constructive monad operator. It is often used as a `return` function at the end of sequences of monadic operations to encapsulate a result into a monad to return it to the calling function. In Haskell the function is usually called `return`.

These then are the basic functions used to build together a particular sequence of monad operations. For any specific monad, new `bind`, `then`, and `return` functions need to be defined. In the case of Haskell I/O, the state monad is called `IO` and so the functions are called `bindIO`, `thenIO`, and `returnIO`.

As noted, `bindIO` and `thenIO` will pass the result of the previous operation as an argument to the next operation; often this will be seen being taken by a lambda abstraction, which in Haskell is written thus `\x -> <expression>`. For use when the previous result is to be discarded, variants on `bindIO` and `thenIO` are provided: `bindIO_` and

`thenIO_`. The underline character is a reminder that each operator is dropping the result normally passed to the right.

These operators just encapsulate the basic structure of monadic I/O. Beyond that is needed some monadic operations which perform specific I/O actions. Quite logically each of the standard dialogue requests has a corresponding monadic operation, which has the same name except with 'IO' appended to it. Thus the examples will be using `readChanIO` and `appendChanIO` for file operations.

### 2.4.3 Monads by Example

In Figure 2.4 is the monadic style version of the example.

```
module Main (mainIO) where

import PreludeGlaIO

mainIO = getinput

getinput   =
     readChanIO stdin                                    `thenIO`
     \input -> how (lines input)

how input  =
     appendChanIO stdout "Do you feel alright?\n" `thenIO_`
     case input of
      (l:input) -> case l of
                     'y':xs -> good input
                     'Y':xs -> good input
                     _      -> bad  input
      _         -> returnIO ()

good input =
     appendChanIO stdout "Great!\n"              `thenIO_`
     how input

bad  input =
     appendChanIO stdout "Sorry to hear that.\n" `thenIO_`
     how input
```

Figure 2.4: Interaction using monadic I/O.

The monadic style shows its colours as a child of continuations here. Indeed the differences between this code and the continuations version appear mainly syntactic and so most of the comments about the continuations example can be carried forward to here.

The general readability is similar to continuations. The program is slightly more verbose with the `thenIO` operations perhaps being more mnemonic than `$`, but in imperative languages where `';'` tends to suffice, this is an arguable point. Writability is improved over continuations. One reason is that each monad operation stands on its own, without complex high-order function to confuse issues, and so it is much clearer exactly what arguments each operation takes, and where they should be.

### 2.4.4 Assessment

Again the problems that the other systems have are used to assess how good the monadic I/O system is for the programmer.

- Modularity.

  The previous problems with lists are not so much an issue in the monadic style. The request and responses lists simply don't make any visible appearance. Whether they actually exist or not is up to the implementation of monads and whether monadic I/O is primitive or built on top of one of the other systems. The input list is still a problem, still being separated from the main I/O system after its creation, introducing coupling between functions as it is passed around. Fortunately, as a result of the extensibility of the monad style this is not a major problem.

- Flexibility.

  As shall be seen in Section 2.5, the monadic style lends itself to being extended by the programmer, in a safe and convenient manner. So a programmer can build up compound monad operations from the simple atomic ones and use them in their programs without extra effort.

- Effort.

  The way in which monads work guarantees that in a expression "`op1 `thenIO` op2`" op1 must return a result before op2 can happen. This means that it is easier to write a program knowing that prompts will appear before input is needed.

  With dialogues, every request has a response which will be handled in a different part of the program. Often these responses are simply a report that no error had

happened. Continuations correct this problem, catching any important responses and passing them as arguments on to the next continuation. Monadic I/O works in a similar way to the continuations style, but with a tighter discipline. All monadic operations return a value which is to be used by the next operation or else discarded by using a `thenIO_` operator.

Unfortunately, the current design of error handling for monadic I/O is not as elegant as for continuations. Presently, the programmer can either ignore errors, which will lead to program termination when an error happens, or else can choose to use an extended monad which also carries error information. This extended monad system allows a monad operation to be specified to handle the error in a similar way to continuations, but in making this choice the programmer then has to change all existing monadic coding to use this extended monad.

The monadic system does not lead to as many type problems as seen with continuations and, as a result, user functions tend to be of simpler types, at least on the surface. In Section 2.5 there is some discussion on some other problems which may surface.

### 2.4.5 A Further Example

The ATM example using monadic I/O turns out to be very similar to the continuations solution. In the same way that continuations needed to have its return type extended, a new monad was required which would allow user input to be carried along with the rest of the program state. It would have been possible to avoid this by using the extended choice of primitives available with monadic I/O, but it was decided to use only the facilities available in continuations to ensure a fair comparison.

There was some initial difficulty building the new extended monad operations, the style being unfamiliar. As with continuations, confusing type errors from the compiler did not help in tracking down these problems, but again, explicit typing brought out the problems more clearly.

Once the new monad was working, implementing the ATM operations in terms of this monad was remarkably straightforward, with only the previous complexity of continuations leading to problems when similar implementation were attempted. Instead, the monadic functions tended to be simpler, with easy types. Unfortunately, the monadic

program does not look as nice as the continuations version, as it has so many ugly infix `thenIO` functions which obscure the code.

The program listing appears in Appendix B.2.

### 2.4.6 Summary

Monads come out very similar to continuations. Their unique feature is the high level of encapsulation which means that all the passing of data between continuations is internalised and the amount of effort required to build monadic programs can be less than for continuations. However, this bundling up of state in the one token introduces a coupling where state is shared between many functions.

## 2.5 Glasgow Monads

As just seen, monadic I/O is very similar to continuations. But their property of encapsulation enables the Glasgow Haskell Compiler designers do some interesting things with them.

### 2.5.1 The Glasgow Style of I/O

The Glasgow compiler uses the monadic scheme as its primitive I/O system, in that when you use the standard dialogues scheme, its lazy lists are interpreted and created by a library function which was programmed using the monadic I/O system.

Glasgow's Haskell compiler implements its I/O system by direct "C-calls" to external libraries, potentially causing side-effects in evaluation, but allowing direct access to all external systems. With this ability, it would be possible to implement dialogues directly within the run-time library, so why does it use monads?

The answer is because with monads there is a discipline of evaluation which would not be guaranteed otherwise. The single-threaded nature of monads means that it is not possible for dialogue requests to be evaluated out of turn or for a response to be made available before the corresponding request is processed.

The monadic I/O system is also made available as an alternative to dialogues and continuations to the normal Haskell programmer, who can now receive its immediate

benefits. Instead of being limited to the small number of primitives provided by the dialogue system, the programmer can use either a (currently limited) standard library of external calls, or write his own C-calls to external systems. As external calls can cause side-effects, it is first necessary that these be sequenced by the monadic I/O systems, and that the programmer has made sure that referential transparency is not lost by any resulting side-effects. If this is not done, the results of executing the resulting programs cannot be predicted.

The library of external functions made available is based upon a subset of what is available from the standard C library, rather than the limited requests from the dialogues system. This has an immediate benefit, that user input can be requested a bit at a time using monadic C-calls to the C functions `fgets`, etc., rather than using the `readChan` request and having a lazy list of characters which would have to be passed around and maintained, even in a monadic framework.

So the functional programmer now has a way of calling external functions which can cause side-effects, while retaining referential transparency.

### 2.5.2 Benefits

There are a number of more direct benefits, both to the compiler author, for whom the monad system was first intended, and to the end user, to whom it is a bonus.

For the implementer, the C-call system means that it is possible to create the whole I/O system using Haskell itself. If it is easier to write parts of it directly in C, then they can still be called from the functional language.

For the programmer, a highly imperative style can be used within a functional framework. This means that translating from C into monadic C-call is fairly straightforward, and could almost be done automatically. This can give you an interesting mix of functional and imperative code.

A more tangible benefit comes from the efficiency of the C-call mechanism. If the compiler is able to unfold the definitions of the monad operations, then the functional program will be compiled down to highly efficient code. In long sequences of C-call operations, the code generated will basically be these C function calls, with none of the standard list-manipulation overhead seen in the dialogues example.

### 2.5.3 Limitations

The monadic system is by no means perfect and there is much research to be done on problems such as the difficulty in handling more than one monadic world in a program [18]. If, using monads, there is a single thread of interaction and a single thread of state transforming functions, it may be desirable for these two threads of operation to work together in sections of the program. Currently this is not possible.

It was noted earlier that C-calls, with monadic control, can lead to greater efficiency, but this system still has difficulty implementing such standard requests as `readChan`, which creates a Haskell channel supplying input in a lazy manner, as it appears. In order to be able to implement this request it is necessary to go below the level of monadic control of the thread of execution and have a concurrent branch of execution involving C-calls which is not governed by the safety mechanisms builtin to the monadic style. This mechanism is also available to the regular programmer, who would be expected to use it with the utmost care as there are no controls on what is allowed at this level.

The need to leave the clean monadic world now and then arises because the monadic scheme tries to hold a complete representation of the outside world within a single token, rather than actually hold the complete state of the world — an impossible task. What is actually required in the case of `readChan` is for the current state of every open channel to be held in the monad. This would lead to implementation problems, and experimentation also shows it to be very inefficient.

Earlier it was asserted that converting from C into monadic C-calls can be fairly trivial. This is at least for straight sequences of function calls, with semi-colons between them. It is hard to match the flexibility of control flow possible in imperative languages even using Monads. For example, all conditionals and "loops" needed to control basic interactions must be performed in the functional language, causing an unfortunate mixing of language styles.

### 2.5.4 Dangers of C-Calls

The "imperative within functional" style can be deceptively attractive and can lead to some ghastly hybrid programs. A common result is a functional program which is mostly imperative code. Out of frustration with the non-global state of functional programming,

```
mainIO =
    malloc 4                    `thenIO` ( \ state ->
    malloc 4                    `thenIO` ( \ flag  ->
    assign state normal         `thenIO_`
    assign flag  false          `thenIO_`

<<application code deleted>>

    free state                  `thenIO_`
    free flag                             ))
```

Figure 2.5: Example of embedded C in a functional language

the imperative code will have its own global memory management system, rather than passing small amounts of data back and forward with a state monad, because this has a high overhead.

Figure 2.5 contains a section of code from an application written as a student project showing heavy use of monadic I/O with C-calls, and acts as a good example of how the C-call system can be abused when writing "imperative" code.

This code does explicit memory allocation and assignment of two variables so that they may be used efficiently in other C-calls within the application. It looks a lot like the code which the programmer would write in C, except that in C local variables get memory allocated on the stack automatically. The corresponding C code consists of two lines declaring and initialising the variables.

This functional code uses none of the features of functional languages, instead it is using the functional language as a meta-language to hold sections of imperative code together. Its programmer is battling against the clean semantics of the functional language to generate state transforming semantics.

Further, the code produced is hard to maintain, because it is hard to read[5] and more prone to errors in programmer-controlled memory management.

[5]The section of code included is remarkably readable compared to other code in this application.

### 2.5.5 Two Way Communication

Monadic I/O provides a convenient way for a functional program to call out to external systems using the C-call facility. But while control of execution is held by an external system, it is difficult for the external system to do the reverse and call code back inside the functional program. Being able to do this would allow an external system to directly control the execution of portions of functional code. The functions called may even jump back out of the functional world via further C-calls, leading to a mixing of calls back and forward between the two languages.

These features would be useful to allow common operations such as mapping external user events into actions on the part of the functional program, or to handle exceptional conditions, such as signals, occurring during the execution of the program. In the Unix process model, signals are a form of interrupt and are used to inform processes of exceptional conditions. They are handled by the operating system calling a program-registered section of code, which would then modify the global state in order to influence the execution of the program. One proposed solution would be to allow the functional program to poll for signals, but often such signals require immediate attention which cannot wait for the program to get round to checking whether there has been a signal, while it was engaged in other activity.

To allow external code to call into functional code, the run-time system of the functional language which would manage this would first have to overcome a number of obstacles.

1. Heap Consistency.

   Functional code cannot be executed if the heap is in an inconsistent state, such as in the middle of a garbage collection. Much of the operation of a functional program is not re-entrant, unless special care has been taken in the compiler to allow this. This means that it is very difficult, if not impossible, to have Haskell code executing during signals. It is only possible when you know that the heap is in a consistent state, perhaps during a C-call, but signals can interrupt execution at any time.

2. Types.

   Haskell types tend to be arranged in memory differently to the types found in non-functional languages. As such, converting between functional representation

of values and non-functional representation can be difficult. Obvious examples of this are representing large record structures, often found in C libraries, in a Haskell data-type. The easiest way to achieve this would be to make the data type abstract inside the functional language, and provide selection functions to manipulate the fields of the record via further C-calls. This is hardly an ideal solution, however.

3. Referential Transparency.

   It is important, when a call from outside the functional language causes parts of the functional program to be executed, that referential transparency is not lost. Thus the functional code called must not directly influence the state of the currently running program. This might imply that such a call is a totally pointless operation; however, the monadic I/O system allows state to be carried along implicitly, allowing modifications of state which can be seen in the rest of the program, but which do not compromise referentiality. So, as long as changes of state occur through monadic operations, referential transparency should be maintained.

Thus there is a limited way for outside code to call back into functional code. External systems, such as window and operating systems, commonly wish to call code supplied by the programmer to handle exceptional conditions (such as operating system signals) and user input (such as window system callbacks, triggered by user events, or from non-user driven external communications), but there is only limited scope within the functional monadic world to cater for this. Exceptional conditions which arrive asynchronously cannot be handled by the operating system calling an arbitrary section of the functional program, as the compiled functional code is rarely re-entrant, and has limited scope if the code cannot modify the global state to reflect its actions. It is only in the case when the functional program has already surrendered control to the imperative world that it could call sections of functional code, passing the global monadic state, without fear of failure due to inconsistent state within the functional world.

### 2.5.6 Summary

The monadic style of I/O is only especially useful because of the C-call feature that comes with it. It gains little over continuations when seen with its disadvantages. Its major

contribution is of controlling the way that C-calls work, to sequentialise them, ensuring that referential transparency is not compromised.

The overall system is very powerful, but the use of low-level calls to C can still be questioned. It is very good at providing a C-style language by which functional programmers may access features of a system not normally available, but if functional programming is to convince people that it really is a step above imperative programming, then such regression is worrying.

## 2.6 Why is it Hard to Interact with Functional Languages?

Functional languages have a problem with external communication. Although dialogues and continuations provide a method of communication, they are rather limiting. Monadic I/O certainly extends the power of functional I/O, but there is a compromise in readability and the solution is not total.

What are some of the problems found when trying to write functional programs that are interactive? I feel that there is something fundamental about the applicative, declarative style of functional languages that makes them poor at creating interactive programs.

In this section I shall review a number of features of functional language, starting first with simple problems, to see what the root causes of this problem are.

### 2.6.1 Modularity

An important concept in software engineering is separation of low-level code, dealing with operating systems, etc, from the higher-level "application" code, which is more concerned with functionality. This allows the low-level code to be modified and rearranged in porting it between systems, without any modification of the higher-level application code.

Currently, there is nothing in the way that any of the standard I/O schemes work to encourage modular programming. The fact that lists, continuations, or monads have to be threaded around a program, introducing couplings, makes it hard to structure code without dependencies between high-level and low-level code.

At the same time, if there is any global state in your program, this needs to be passed around between many functions that are not interested in the contents of the state. Further, this pseudo-global value has be declared in one place and is visible within all functions through which it is passed. This makes it difficult to create truly modular programs.

### 2.6.2 Multithreaded Input/Output

Continuations and monadic I/O have at their heart the idea of a single thread of operation: the reason for this is mainly to ensure that state changes do not break referential transparency. Single threadedness is unfortunate, in that interaction tends to be a multithreaded activity, with users likely to be switching their attention between various parts of an interface. If the functional program can be carrying out only one piece of dialogue at a time, as in the ATM example, then the user is tied into a single thread of interaction. Either the single threaded nature of functional I/O has to be multiplexed (as in the Fudgets system presented later, in Section 2.8.2), or perhaps Haskell's I/O system needs to develop some way of achieving multithreadedness, perhaps using the ideas that Holyer and Carter propose [14]. Their proposals are discussed more fully in Section 3.1.2.

### 2.6.3 State in Functional Languages

As noted in the introduction, there are some non-pure functional languages for which interaction with external systems is not a problem. Could it be that there is some inherent property of pure functional languages which means that interaction will always be a problem? The very nature of interaction is based around the modification of shared state through an interface. Unless the language is tolerant of these side-effects in some way, then state changes as a result of external systems will always be hard to control.

In current systems, in order to control the handling of state, lists or tokens need to be managed or, in the case of primitive continuations, it is necessary to work in an awkward higher-order style, where program readability can be compromised. Certainly, writing programs in any of these styles can be a trial.

The clean management of state in functional languages is difficult, while in imperative languages it is simple. The imperative language's semantics are concerned with a global environment, with some scoping rules and, as such, directly addresses state changes.

The functional language's semantics is based more on manipulating values, and does not address global state well at all.

### 2.6.4 Summary

There are a number of specific reasons why functional languages are poor at interaction. Firstly, functional languages do not encourage modular programming, which makes some aspects of interaction difficult. Secondly, the single-threaded nature of I/O in functional languages is at odds with the multi-threaded nature of the external world. Thirdly, as interaction tends to be based on modifications to an interface shared between the user and the program, the apparent statelessness of functional languages does not help in programming interaction. Lastly, functional programming has not yet found a good way in which to separate application code functionality from interface functionality, making it difficult to produce well-written interactive programs.

## 2.7 Further Requirements

By investigating limitations in current solutions for simple I/O in functional languages some more specific requirements are found which any solution to the problem of interaction with functional languages needs to address. Some of these come directly from the discussion of the monadic style.

### 2.7.1 Modularity

It is common for interaction code to become entangled with application code. I should avoid this in any system I build.

This requirement links back to the original requirement that the interface be separate from the application, but with an added fact that doing so will make programs more modular.

### 2.7.2 Flexibility

The traditional dialogues and continuations schemes are non-extensible, meaning that only the operations determined by the language designers are available. Monadic I/O

does not suffer from this problem, nor should any future solution.

### 2.7.3 Effort

Some of the programming required to achieve the most basic interaction is quite detailed. More so, a lot of it is fluff, dealing with verbosity of the I/O system, or awkward error handling. Often it is hard to see the useful code amongst all the bits of irrelevant code holding it all together. I want a more direct style of coding. Notice that the monadic style has its own limitations, and it is not a complete solution to interfacing functional languages to external systems.

### 2.7.4 Multi-threading

User interaction can have a high level of concurrency, and it is apparent that, currently, functional languages make it difficult for different logical parts of a program to be able to take part in interaction without a great deal of cooperation between the various parts, as they pass around a monad or manipulate the lists of the dialogues system. I wish to be able to manage interaction within independent threads of a program.

## 2.8 Other Proposed Solutions

As user interaction within functional languages is not a new problem, there have been many solutions proposed. In the introduction I covered some of the history of my own work and now I shall cover some of the solutions proposed by others.

Many people have written of the problems functional languages have with user interfaces and have proposed various solutions. These solutions range from some of my own simplistic solutions [32] to the more powerful systems that Singh has produced [33], with some truly innovative possibilities explored by Carlsson and Hallgren [5] and by Dwelly [8].

Because I wish to develop a system for building interfaces outside the functional language, it is not necessary to examine every system which allows the building of user interfaces inside the functional language, but will concentrate instead on two currently popular schemes. The investigation begins by looking at the research area in general, examining in particular schemes which also employ separation.

### 2.8.1 Overview

All systems built to allow a functional language to interact with the user have, at some level, the functional program communicating with an external agent, receiving events from the user and replying with requests telling the agent what to do next. This can be done either by low-level calls embedded within the run-time system of the language, or by an external process, connected to the input and output streams of the language. This difference of technique 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.

Where designers have used a separate program to build and control the interface of a functional program, it has always been that this process had no intelligence of its own. The functional program was, in effect, controlling a robot which would create the interface, but could only channel user feedback directly to its controller, not being able to decide for itself how to react. This means that although there is a physical separation of interface and application, there is no separation of control, everything being managed in the functional program. This solution works, but is certainly not elegant.

To my knowledge, no one else has proposed using an programmable agent to control the interface for functional programs, taking away the effort of programming interaction functionally. However, the idea of a programmed agent is not new, its use outside the functional world implies that there must be some merit to the idea.

### 2.8.2 Fudgets

A recent innovation in the field of graphical interaction and functional languages is the Fudgets system, by Magnus Carlsson and Thomas Hallgren [5]. This is a complete window system toolkit written in a lazy functional language, making heavy use of higher-order functions to provide its power. This is the first full implementation of a graphical interactive toolkit in a lazy functional language.

#### Review

The Fudgets system is very impressive. It allows some very complex programs to be created without leaving the functional language. Its authors have created a number of

relatively large demonstration programs, showing the flexibility of the system.

The key concept is a so-called 'functional widget', or 'fudget'. Fudgets correspond to the widgets of imperative toolkits, providing buttons or simple dialogs which, when combined, build into a complete program. Functionality is provided either directly in the semantics of each fudget, or else by non-display fudgets used to build other fudgets together.

The authors acknowledge the desire to separate the actions generated by user interaction and the resultant processing of those interactions. Indeed, they achieve this aim quite well, but it appears that the structure of the application is tightly bound to the structure of the high-level interaction. This means that it would be difficult to redesign the interface to one of their programs without some alteration to the structure of the application.

There are some other, less fundamental, problems with the current Fudgets scheme which I summarise below.

- Error handling.

  Currently there is no discussion of how errors returned from the window system may be handled. I can only assume that the system always take the common route of exiting when such a fundamental error occurs. This is forgivable in an experimental system, but is unacceptable in a production system.

- Run-time tailoring.

  It is possible to set "resources" (configuration options) for individual fudgets when they are created, but there is no obvious way of modifying these resources during the lifetime of the fudget. This makes it impossible to manipulate the interface during execution, and also limits the over-all flexibility of the system.

- Information flow.

  There is a limited path of information between the application and the interface. All control and information paths between application and interface have to be explicitly programmed. To modify an already programmed application could require extensive rewriting of the interface, when the current paths of information flow need to be altered.

- Efficiency.

  The Fudgets scheme makes heavy use of lists. The threads of I/O that the fudgets process are initially sourced from the Haskell I/O replies stream, which has to be split up and routed appropriately, and channeled into the requests stream in an elaborate multiplexing scheme. Thus, there is a lot of list construction and destruction within any fudgets program. This could be addressed by a deforestation optimisation, but such optimisations are not common in real compiler implementations.

- Program Structure.

  As discussed in Section 1.2.3, there is a multi-threaded nature to user interaction and, indeed, fudgets reflect this in some sense by having each 'fudget' as an individual thread, which is run independently of other fudgets. These fudgets then need to be connected by making explicit the dataflow between the program and its interface. This can impose an inelegant style of programming within the application core. If the flow of data in the interface differs greatly from the dataflow of the application then coding can become difficult. Further, changing layout may require major changes to the structure of the application. This is very poor in modularity terms, as there is a very tight coupling between interface structure and application structure.

- Locality

  Each fudget, unless specifically controlling a number of sub-fudgets, has no access to the state and resources of other windows. This makes cooperation between fudgets difficult, such as a fudget controlling a palette of tools which needs to be in a separate window from the fudget where the tools operate.

#### A Different Implementation

Alastair Reid and Satnam Singh have developed an implementation of Fudgets based upon the OSF/Motif widget set [22], using the monadic I/O system. This system has an efficiency gain over the original fudgets system since much of the tedious list construction and destruction has been replaced by direct calls using monadic I/O.

**Summary**

The Fudgets system is a good way of structuring applications, given that application dataflow is unlikely to change especially radically during the life of a program. However, building an interface above this structure is not always good because there is too tight a coupling between application structure and interface structure which can affect future maintenance.

### 2.8.3 Concurrent Clean

Concurrent Clean [10] is a lazy functional programming language which runs on a number of different platforms and features concurrency operators and extensive I/O abilities. To ensure good portability between platforms, Clean implements its own abstraction of the various I/O facilities, avoiding the need to call outside the functional world and thus the problems such calls create. A detailed description of how Clean I/O works is in [2].

**Review**

Like Fudgets, Concurrent Clean does its graphical interaction by imposing a particular style of coding. In Fudgets the code is structured by the dataflow of the application. In Clean the structure is by a hierarchy of event types.

Windows and menus are registered as interested in particular types of events with a main event dispatcher, so that the appropriate block of code is called when particular events happen. These functions can then do output operations by using the built-in abstract functionality of Clean.

The Clean I/O system, because it uses built-in datatypes and functions to achieve its aims is, unfortunately, hard to extend in comparison with the monadic system. Unless a rich enough set of I/O primitives have been provided, there will be programs which just cannot be written in Clean. As the authors of Clean claim their I/O system to be at a very high level of abstraction, it is questionable whether enough low-level systems to ensure flexibility is provided.

Clean provides functions for all output operations. In order to achieve this without compromising referential transparency, Clean requires a system which achieves the same as monadic I/O; i.e. which avoids side-effects to shared data. Monadic I/O does this by

encapsulating all mutable values in an abstract token which cannot be shared. Clean does this by extending the type system, introducing values with "unique" types, which the type system then ensures that these values are only used linearly, i.e. that they are not shared. In practice, the restrictions imposed on the Clean programmer are not especially different from those imposed upon the monadic programmer, with the type system implementing the discipline for both regimes.

Alas, Clean does not live up to its name. Either its syntax, or the coding style required by the I/O system, renders programs written in Clean as rather opaque. This is unfortunate as it is becomes hard to compare programs written in Clean with those written using either Monadic I/O or the Fudgets system.

### 2.8.4 Summary

There are no complete solutions to the problem of creating interactive programs with functional languages. Both Fudgets and Concurrent Clean attempt to solve the problem from within the functional language. In both cases the result is extra effort in programming, with reduced efficiency, reduced clarity and reduced extensibility being related problems which affect one or both of the systems.

## 2.9 Summary

I have looked at the current standard techniques for programming input/output within Haskell. This comes down to a choice between dialogues, continuations or monadic I/O. Monadic I/O is currently gaining favour, and has removed some of the limitations in the other systems. Dialogues-style I/O has been shown to have little value beyond very short programs, and its use is not recommended. Continuation-based I/O is, at its heart, very similar to monadic I/O and could achieve similar results if it were given the same level of consideration as is given to the monadic style.

With all this said, I still find that functional programming language, complete with their modern I/O systems, are still quite inadequate for writing interactive programs. In systems such as Fudgets, or Clean, where the language has been extended, a solution has been searched for within the functional language and the result has been compromises and complexities. There are fundamental problems in the functional style which limit its

usability in interactive systems.

# Chapter 3

# Making Interaction Easier

It is now clear that there are problems when functional programs are asked to communicate with the non-functional systems which allow interaction with users, and that none of the systems considered so far solve these problems. In this chapter I shall explore these problems further, proposing a solution based on ideas seen in Section 1.2, taken from software engineering, HCI and UIST research.

A valuable principle of UIST is that the user interface should be separated from the application core. I shall show see how this can be achieved in functional systems, using the Tcl/Tk toolkit introduced in Section 1.2.4, and how this solves some of the problems listed in Section 2.6.

By extending Tcl/Tk, I shall show how it is possible to reduce the effort required to produce good user interfaces from functional languages, providing greater ease-of-use than given by the monadic I/O system. The resulting system will be extensible, so that it can grow with the programmer's needs, and which will integrate smoothly with the functional language, not disturbing the smooth functional programming style.

To illustrate my solution I present four example programs — two new ones written to demonstrate the practicality and functions of my design and a further two based on the examples from the previous chapter. I shall also examine possible further developments, and different approaches to the development of my system.

The interfaces are written using the Tcl/Tk toolkit introduced in Section 1.2.4. By moving interaction with external systems from the functional program into a separate process which communicates with the program at a high level of abstraction, I improve

maintainability by increasing modularity, portability and coherence of software design. The interface and the functional program communicate by means of a high-level protocol, defined by the program designer. A good choice of protocol should reduce coupling between interface and application core.

I have gained an immediate improvement on effort required to build interfaces by using Tcl/Tk. Its efficient, clean syntax makes interface design, creation and modification a pleasure, since it is easier to program than even the normal imperative languages, and also more straightforward than the use of the C-call feature of monadic I/O.

I start by reviewing Tcl/Tk, considering how it should be extended.

## 3.1 Modifications to Tcl/Tk

In Section 1.2.4 Tcl and Tk were introduced. Tcl is a simple embeddable language for which a graphics user interface toolkit called Tk has been created which uses the Tcl language as its core. Using Tcl, scripts are written to control the actions of the toolkit.

The Tcl interpreter used by Tk is made available in a program called `wish`, standing for *wi*ndowing *sh*ell. It is common to extend the Tcl/Tk system with new features, and create an enhanced version of `wish` with a different name. I have followed this lead, creating an enhanced `wish` called `swish`.

I have extended the Tcl language with commands to allow communication between Tcl/Tk and an external system which, in this case, is a Haskell program. This allows programs written in functional languages to communicate with users through interfaces written in Tcl/Tk. The design and implementation of this is discussed below.

### 3.1.1 Design Background

There is a long history of grafting user interfaces onto the front of less interactive programs. These normally work by creating 'pipes' of input and output for the core program to which the user interface is attached.

This approach has also been used to create user interfaces for programs written in functional languages [32]. However, this approach lacks something in that either the interface has to be tailored for the individual program or an elaborate protocol has to be created which allows the functional program to control the creation and running of the

interface that the front-end program creates [33]. Again, the mechanism, which allows complete control of the interface to be dictated by a separate communicating process, has been implemented in the non-functional world to permit programs, written in languages without the flexibility to call C functions, to have graphical user interfaces. A good example of this is a small program called 'dox' which is designed to allow Unix shell-scripts access to Xlib functions. Naturally, the programming language used with a system like 'dox' could just as easily be functional.

Another program in a similar vein is 'Wafe' [20], whose purpose was to allow graphical user interfaces to be created for programs which cannot interact for themselves. The major difference between Wafe and 'dox' is that with Wafe the interface is programmed in Tcl. This allowed interfaces to be designed in Tcl, a language well suited to interface creation and management, allowing whatever program it attaches to, to concentrate on the application functionality.

Wafe can be used to create user interfaces for programs written in a functional language. In [31] I created a couple of examples of functional programs using Wafe for their user interfaces. However, Wafe has a number of limitations (e.g. no ability to do direct graphics) and, in my opinion, design flaws (its method of attaching the two processes together is limiting). This led me to an alternative tack.

Being impressed with the Tcl language that Wafe used, I decided to investigate Tcl/Tk, the Tk toolkit having features lacking in the Athena toolkit used by Wafe.

It was at this point that I chose to use Tcl/Tk, but any other toolkit or UIMS could have been used. One particular reason for my choice was its nature as both a language and a toolkit. Using a toolkit with a built-in high-level interpreted language allows a programmer to adapt the interface at run-time from the functional programs if desired. This allows tailored code to be loaded into the interface, and thus ensures that it is always running at its most efficient.

### 3.1.2 Process Communication Design

In current systems functional programs have to communicate with external systems at a relatively low level of abstraction, directly using whatever I/O system the language provides. I want instead to have the functional program communicate with a interface

broken off into a separate system, and communicated with at a much higher level. Such an interface would need to be specially written for each particular application, and it would be programmed to do all the low-level interaction that the functional program would have had to do otherwise, communicating with the functional program at a high-level.

Getting the communications right is essential. A poor choice of design for the connection of interface to application, risks making it difficult or impossible to code some interactions. If the wrong level of abstraction is chosen for the communication, then there is the risk of over-loading the channels and creating a bottle-neck or, at the other extreme, limiting the expressive power of the communications.

The interface and the application communicate using the standard I/O system of the functional language, and whatever I/O system is available in the interface's coding language. The two parts of the program do not have to be separate processes in operating system terms. It is possible for the two parts of the program to exist as cooperating threads of execution, where this can be implemented within the language. It is then possible to use call-backs to channel information from the interface to different parts of the application. Also possible is true concurrent threading, if the operating system supports it within the same process.

Assuming that the interface runs as a separate process from the application, it is necessary to look at how the interface and the application will communicate. An obvious first idea is to have two communication channels between the interface and the application; this is how 'dox' and Wafe work. Then all messages the interface generates may be passed on to the application, and any responses can be sent back in the other direction from the application. User events will also be sent along this channel, asynchronously, to ensure they get timely processing. This system is illustrated in Figure 3.1.

I reject this model for the following reason: if the application has to make a query to the interface, it then has to find the reply to the query within the input stream arriving from the interface, while ensuring that incoming events are not lost. How hard this is to achieve in functional languages can be seen in the implementation of Fudgets [5] where special handlers are required to queue events while searching for replies awaited. The same queuing and filtering process can be seen in the protocol of the X Window System [29], where replies to X protocol requests and user events are multiplexed on the same connection and must be carefully separated and requeued as appropriate.

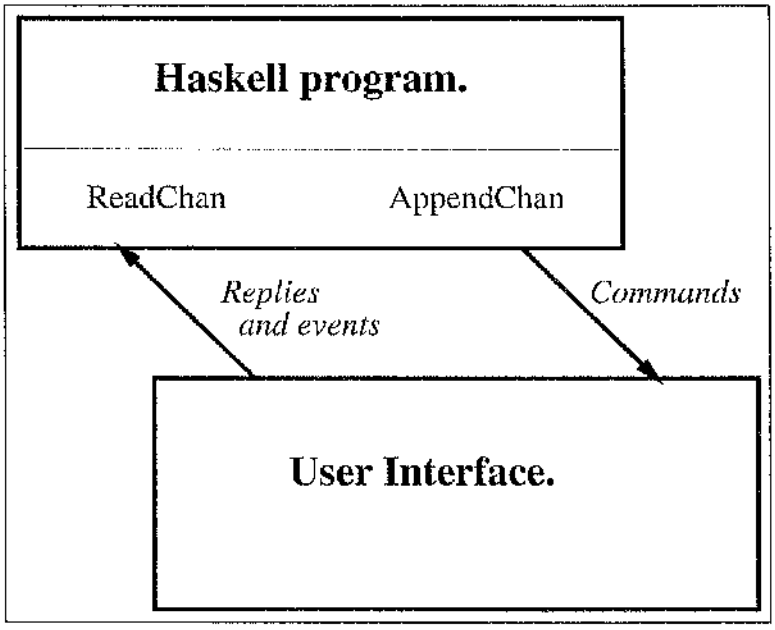


Figure 3.1: Two-way communication between Interface and Application.

Ideally there would be four channels, so that each system has both an asynchronous and a synchronous connection to its peer. This would allow easier handling of incoming communication, but with the overhead of twice the number of channels of the two-way system. Is there a real need for this amount of bandwidth to be maintained and controlled? If X can work, albeit with some effort, with only one bidirectional connection, perhaps four unidirectional channels is more than required.

Between these two extremes I favour a three channel solution, as illustrated in Figure 3.2. From the interface to the application there is firstly an asynchronous channel for events. Events can be sent on this channel at any time. From the application to the interface is a command channel. A second channel from the interface to the application then allows synchronised replies to commands sent on the command channel. The Haskell application can handle concurrent reading from more than one incoming channel, if necessary, using the `ReadChannels` primitive, rather than the simple `ReadChan`.

Some early results have shown this model to be much easier to program than the previous norm of two channels, which led to an almost synchronous protocol between the cooperating processes to ensure that events did not get intermixed with other communications.

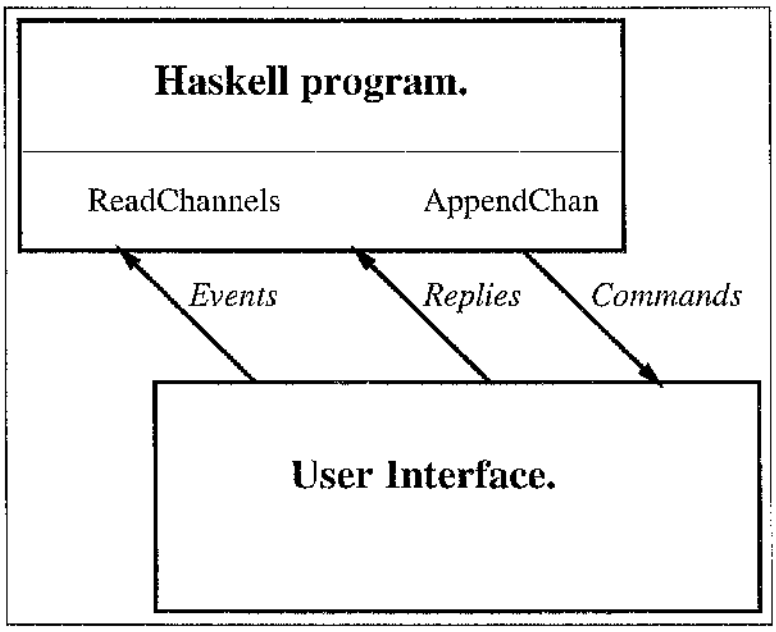


Figure 3.2: Three-way communication between Interface and Application.

This three-way system bears some similarity to the system of communications devised for Took's Presenter system [36]. Presenter is a "surface interaction" system, which, in place of a simple window system, provides an abstraction of windows, frames and other interactive objects with which interfaces can be built. As in my system, the presenter system is a separate program which communicates with the application to provide its services. Between the presenter system and the application it is serving, there are three communication channels established, which correspond almost directly to the three channels I employ.

### High level interaction

Naturally, the functional program will still have to communicate with the outside world, but via the separated interface. This will have to be managed using the traditional I/O system of the language. The difference is that that the functional program will now take part in the interaction at a much higher level and so more information can be communicated, thus increasing the bandwidth. Instead of passing the complete dialogue needed to achieve a particular aim, the functional program only has to communicate the intention, with any necessary data, and the interface can look after all the details. Equally,

the information returned from the interface to the program could be at a much higher level than that found in typical I/O systems.

For example, if the user wanted to save a current document, the discrete events generated by the user would not be seen by the application. Instead it would be handed a high-level event conveying the user's wish to save the document. At this point the application would pass a copy of the document to the interface for it to save to disk, without the application needing to go through the step-by-step processes involved.

So there is now a separation of the low-level processes which happen in both the interface and application and the high-level dialogue which the two sides enter into. At the lower level is key presses and mouse click in the interface and calculations and manipulations in the application. The high-level dialogue does not deal with such details.

**Protocol Format**

Communication protocols tend to be based on low-level byte formats for efficiency. For the purposes of this exercise, however, efficiency is not a major requirement and the desire for an easy interpretation of the communication streams leads me to choose a flexible text based protocol, where each line of text (terminated by a new-line) is the basic packet.

This allows the functional program to communicate with the Tcl/Tk process on the command channel using simple Tcl commands which can be handled directly by the Tcl interpreter. For communication in the other direction, Haskell has flexible routines for handling streams of text, and will allow direct pattern matching of words. An additional benefit is that a textual protocol is easier to debug by inspection or mimicking that a byte-stream.

So events sent from the interface to the application will be simple strings representing actions on the part of the user. Examples might be "quit," to terminate the program, or "font 12," to set a particular piece of text to a different font size.

In reply the application will send strings to the interface. Examples of this might be "about," which would produce an 'About' window for the program, or "change old-string new-string," for application-led manipulation of some data used in the interface.

#### The Application Core

I have specified how I would like to construct the interfaces, using Tcl as a high-level protocol to afford communication with the interface. Now I consider the remaining application core of the program. What particular features are required within the application, and how the application is structured must now be considered.

The monadic system of embedding C code inside the functional language, as shown previously, can lead to ugly code. By having a separate interface which will contain the previously embedded imperative code, the purity and coherence of the functional code can be improved. It certainly makes more sense to write imperative code in an imperative language, and functional code in a functional language.

#### Multithreaded Application

To allow complete user freedom, components of the interface should behave in an independent manner with a dialogue in one window not interfering with a dialogue in another. For this reason I choose to have a system of processes, rather than the sequential solutions normally made available by functional languages.

Even though functional languages are intrinsically parallel in nature, there is little notion of threading seen within their design. Again imperative languages come to the rescue. It is now normal for languages, or the operating systems they run on, to offer some sort of threading support [35].

More recently, Hoylier and Carter have proposed an extension to Haskell which allows easier threading by permitting multiple output streams using a new I/O request 'WriteChan' which would split off a separately evaluated stream of output, in the same way as it is currently possible to have multiple input streams with the 'ReadChan' request [14].

### 3.1.3 Process Communication Implementation

Implementation of the process communication system required two pieces of code to be written. Firstly, Tcl/Tk was extended to allow it to set up the two processes of the interface and application. This produced the extended `wish` shell, `swish`. Secondly, the Haskell run-time system had to be adapted to allow it to communicate on the channels

created by the Tcl/Tk system.

### Overview

For simplicity I chose to implement the process communication using the `pipe` system call, as opposed to the more powerful but more complex `socket` operations. Whereas sockets allow bidirectional communication between independent processes anywhere on a network, pipes provide only unidirectional communication and only between processes which share a common ancestry — the pipe is created by and inherited from a common ancestor.

I use three pipes, corresponding to the three data paths in the design. In order to arrange "common ancestry" I have the Tcl/Tk interface create the pipes and then spawn the Haskell program. The spawning is done using the `fork` and `exec` system calls.

This is all coded as an extension to the Tcl/Tk system rather than the Haskell system, as the Tcl/Tk system is smaller and simpler. Doing this also allows me to use the same Tcl/Tk system with different Haskell compilers, or indeed different languages.

The additions to Tcl/Tk are covered below in more detail. The code can be inspected in Appendix C.1. The processes required to create extensions to Tcl and Tk are covered in detail in Ousterhout's book on the Tcl and Tk system [25].

### The Spawnchannels command

The `spawnchannels` command is the main extension to Tcl/Tk. All process and channel setup is done by this command. It has a very simple syntax.

```
spawnchannels program [arguments]
```

The command takes a single argument which is the name of the Haskell program to spawn, plus optionally any arguments to pass to the program when it is executed.

The implementation of the command is fairly straightforward. Ignoring error checking, it simply creates the pipes for the communication channels, then forks. At this point there are now two processes running. One will stay as a Tcl/Tk process, the parent process, the other, its child, will become the Haskell program. These two processes are now covered in detail below, separately.

The parent process after the fork has a simple task. By using the dup2 system call, it arranges its end of the pipes to be at file descriptors 13, 14 and 15 and closes all its original references to the pipes. It does this to ensure that the processes exit cleanly when either side closes a pipe; it is the last close on a pipe which causes it to shutdown, and so it can only exist in one place. The parent then sets these new descriptors to non-blocking mode to guarantee that the interface will never block while waiting for the Haskell program. It finally sets up the handlers for each of the channels as will be described later.

The child process follows a similar path, using dup2 and closing the original pipes. It does not set its ends of the pipes to non-blocking as the Haskell program it is about to execute may not be able to cope with this, and it is not a problem if it blocks anyway. Finally, before execing the Haskell program, it closes the connection to the window system that it inherited from its parent, but which cannot be used in any other process.

Figure 3.3 shows the arrangement of pipes set up by the two processes, showing the number of each file descriptor, and the direction of each pipe.

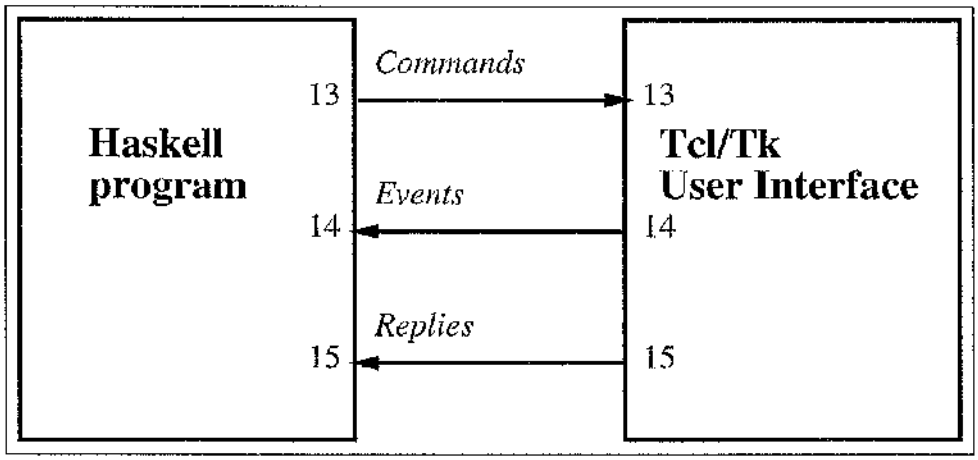


Figure 3.3: Communication between Tcl/Tk and Haskell

At this point, the Tcl/Tk process — the parent process — is ready to run its interface, accepting commands from the user and from the functional program through the command channel.

### The Command channel

The command channel is implemented by the `ComProc` procedure in the `swish` program. It is based upon the standard input handling function, but with more error handling, and some changes of behaviour in exceptional circumstances.

Lines of text incoming on the command channel are assembled into complete Tcl commands — employing the parser to decide if the command is complete or not — then executed. Making use of functions supplied by the Tcl parser to determine whether a command is complete or not avoids the additional complexity that would otherwise be involved in supporting multiline commands.

The `ComProc` procedure is registered with the Tcl/Tk system so that it is called whenever any input appears on the command channel. This ensures that the input is handled in a timely manner, and that the program will not 'hang', waiting for input from the functional program.

### The Event and Reply commands

The `event` and `reply` commands are further extensions to Tcl. These commands send tokens of information from the interface off to the functional language. The danger of blocking is a greater problem. If the functional language is busy processing other data, then the interface must not be delayed.

This is a very real problem when the functional program is acting as a computational engine, and thus would not normally be very responsive. Of course, on the occasions where this problem can be expected, care should be taken within the interface and program to slow or stop communication while the application is busy. The easiest way to achieve this is to lock-out certain parts of the interface while processing continues.

I solve the problem by having a simple queuing system, where any data that the application is not ready to receive from the interface is appended to a queue. Two queues are arranged, one for each of the event and reply channels. Both are important as the functional program may not be processing events when it is in the middle of a command/reply dialogue with the interface. At the same time, unexpected replies will not be read by the program until it is looking for a particular reply to a particular query. (It is for this reason that I would suggest that replies are tagged in order to ensure that

the correct reply is identified.)

Special care is taken to ensure robustness in the face of extra long messages, where dynamically allocated data structures used to hold the messages are extended as needed. Likewise, the queue can grow to effectively infinite length, limited only by memory. It is hoped that the queue would not need to be particularly large and, as security against programs with bugs in them, it might perhaps be better to limit the lengths of the queues to stop large amounts of data from building up in a queue.

When either queue contains data to be sent, Tcl/Tk is instructed to call a function to transfer the data when the channel is ready, similar to the way in which Tcl/Tk calls a function to manage incoming data on the command channel. This ensures that data is sent at the earliest possible opportunity, while ensuring that the interface can never hang, waiting for the functional program.

**Interprocess Communication with Haskell**

The `spawnchannels` command in the Tcl program is responsible for starting execution of the Haskell program. It inherits from its parent — the interface — three pipe ends on which it will communicate. These are found on file descriptors 13, which is used for sending commands to the Tcl program; 14, which is for receiving events sent by the `event` command in the interface; and 15, which receives replies to commands sent on the command channel generated by the `reply` command in the interface.

In order to be able to read from the event channel, the reply channel, standard input, or any other channels the programmer may be interested in, it is a good choice to use the `ReadChannels` request (or continuation equivalent) to read from multiple channels simultaneously, if the particular Haskell implementation provides it. This is especially useful with the Haskell B implementation, which has two pseudo-channels called `TICK` and `TIMEOUT`. These allow you to use `ReadChannels` and, at the same time, be able to perform other actions if user input is not received between ticks or before a timeout has been reached.

The `ReadChannels` request takes a list of channel names which it uses to associate Haskell channels with operating system channels. In most Haskell implementations, the only recognised channels are `"stdin"`, `"stdout"`, `"stderr"` and `"stdtty"`. Haskell

B originally interpreted any other string as a filename to open.

To open actual Unix file descriptors it was necessary to modify the Haskell B run-time to recognise the small positive integer values of file descriptors, and use them when possible. These modifications were fairly trivial.

Thus, using `ReadChannels`, the Haskell programmer can set up a high-level event loop, in the style of any other interactive program, but at a higher level, which can call into functions which can communicate using the command and reply channels. As an output channel, the command channel would be accessed via `AppendChan` requests, while the reply channel, as it is not relevant inside the event loop, would be attached to by a `ReadChan` request.

## 3.2 Two Examples

I present two example programs written in Haskell, using Tcl/Tk as the interface system. The first, a simple clock, demonstrates how it is possible to write programs that can both respond to user input, and update the display at regular intervals of one second. I then present a larger example of a three dimensional maze simulation. The Haskell code involved in this program is quite complex, but very little of it is concerned with dealing with user interface actions.

In this section I only describe the external design of these programs; in the section following their internal construction is examined.

A later section will examine the examples from Chapter 2, where I have written new graphical interfaces for these textual interactive programs. The first of these is relatively small, so is the only one which will have sections of its code presented in the text, rather than in Appendix app:easier.

### 3.2.1 An Alarm Clock

For the first example, a simple alarm clock program, the Haskell program keeps note of what the time is and when it should activate an alarm. Every second it advises Tcl what the time is, using a procedure defined in the script that the `swish` interpreter 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.

When the alarm is activated, the 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.

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

### 3.2.2 A Maze Game

As a more substantial example, I 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 (crawling up walls or over precipices as they are met.) 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 player is facing. When a button is pressed by the player, the program passes on the appropriate event to the Haskell process. The Tcl/Tk program 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. The only constant factor is the protocol between them.

Figure 3.4 shows what the maze program's user interface looks like. A complete copy of the source may be requested from the author using electronic mail. Highlights of the

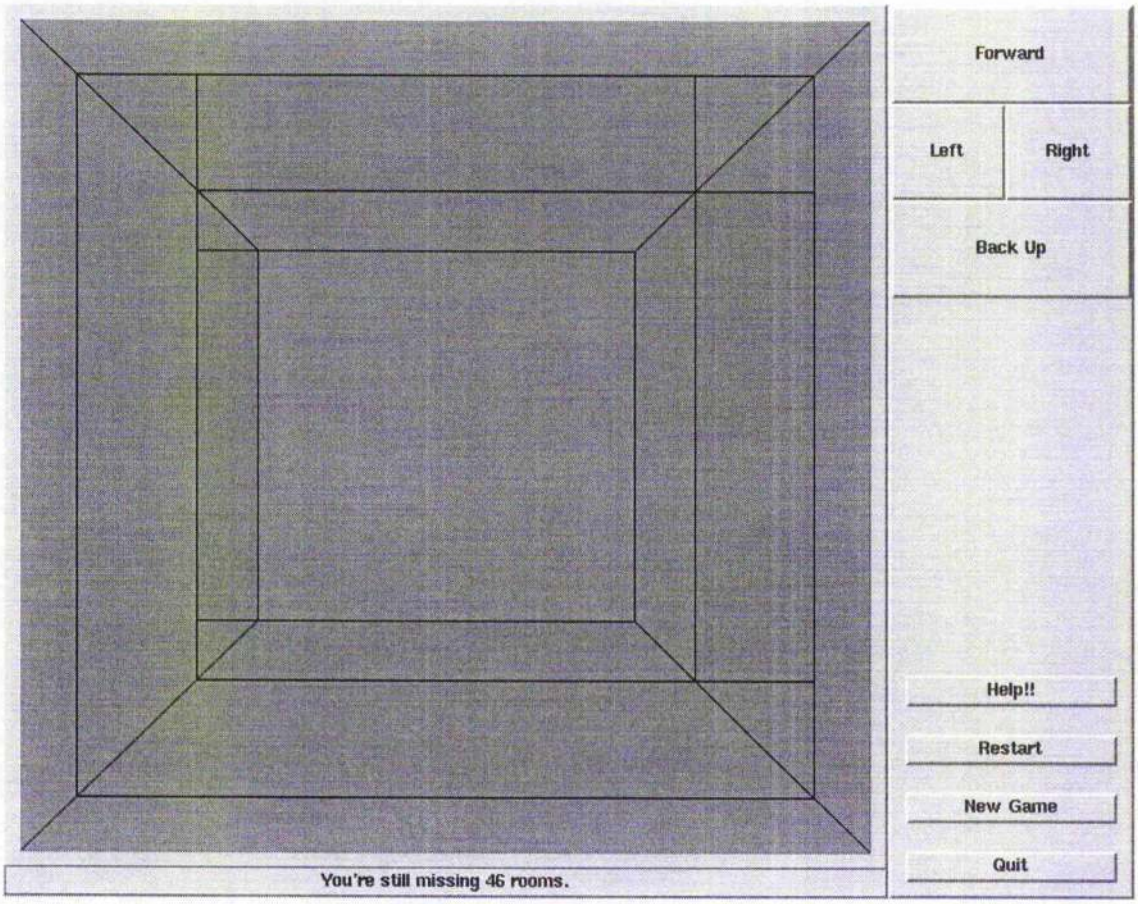


Figure 3.4: Functional Maze in X

Tcl interface code are in Appendix C.4. Listings of the Haskell modules are given in Appendix C.5.

### 3.2.3 Summary

I have shown that it is possible to create useful interactive programs in Haskell, using Tcl/Tk to create the interface. The resulting programs are flexible and responsive. It is interesting to note that the maze application and its interface were developed completely independently.

## 3.3 The Protocols, Interfaces and Programs

The most important part of the creating of these programs was in the design of the protocol used by the application and interface to communicate. The interface builds high-level events to send to the application, while the application will send commands to query or update the display.

As explained above, the interface is responsible for all communication with the user and translates task-specific user actions into high-level events for communication to the application core. These are then sent textually to the core using the `event` channel.

In turn, the interface makes available a number of procedures which the functional core is expected to call using its `command` channel to the interface. Of course, the core may also use any other standard Tcl command, even creating new procedures, or re-writing existing ones if required. This could be useful in a highly interactive interface where display parameters can be computed on the fly using core-supplied functions while still in the context of the interface. A good example of this might be a program which displays mathematical functions selected by the user. Another example is when a data structure controlled by the core also has to be displayed by the interface. The core would mirror the relevant parts of the structure within the interface for rapid display.

In this section I shall examine the construction of the interfaces of the two example applications, and examine the communication which occurs between each and its application core. I hope to demonstrate that the designs meet the requirements laid down in Section 1.3. This will be discussed further in Section 3.5.

### 3.3.1 The Alarm Clock Program

The alarm clock has a simple design with a resulting simple protocol, interface and program. The majority of this program is presentation: not much programming is required to know what time it is.

#### The Alarm Clock Protocol

The protocol used by the alarm clock program is very simple in design.

There is only one event the interface can send to the application.

- `HH:MM:SS`

  A specification of when the user wishes the alarm to be activated, in a strict 24-hour format, using 2 digits to express each of the hour, minute and second. An example would be "`23:59:59`".

There are two commands which the application will use in the interface.

- `disp` *time-string*

  This command should be called periodically to update the displayed time to that given by the *time-string*.

- `alarm`

  This command should be called when a previously set alarm time has arrived.

The replies channel is not used and the information flow across the event and command channels is very simple in form and content.

#### The Alarm Clock Interface

The interface, despite the simplicity of its task, has very little knowledge of its purpose. It is limited to the fact that the dialogues have titles to say it is an alarm clock program written using Haskell and that, within the alarm-setting dialogue, the user should type a time of day. These are the only two strings in the interface of any substance. There is also the flashing alarm dialogue, but this could be used by the application to display urgent error conditions, and again is in no way tied to the operation of an alarm clock.

There is only one event which the interface will send to the main program; this communicates what the user has asked the alarm to be set to. This format of the event is a string strictly of the form "HH:MM:SS", with each component being two digits. This is basically the string that the user types in, with first a validity check to ensure it conforms to the correct syntax. It is important that the interface implements this check as, by the time the application is in the position to check its syntax, the dialogue will have been dismissed.

The code for the interface is in Appendix C.2.

#### The Alarm Clock Application

The main program of the alarm clock has two main roles. Firstly, to update the time display every second and, secondly, to store the alarm clock setting and activate the alarm when the appointed time arrives. The code for the application is in Appendix C.3.

Control of the interface by the program is through two simple commands. The first, `disp`, sets the display of the clock and will be called once a second to keep the clock display correct. The second, `alarm`, is used when the alarm is to be triggered. Once triggered, the application program takes no further interest in it.

The initial implementation of the clock used the dialogues style of input/output but, as I developed it adding the alarm feature as an extension to the original code, I found dialogues difficult to work with, requiring non-localised changes in the program to add the new feature. It was for this reason that I re-wrote the program in the continuations style, again developing the clock portion of the code before adding in the alarm feature. This time the alarm was easily integrated, requiring only one new function, and one other function to be changed to call the new function at each tick.

### 3.3.2 The Maze Program

The maze game was an idea by Carsten Kehler Holst, and he agreed to write the main program. As he was in Sweden at the time the program was written, it was vitally important the the communication protocol between the application and interface was clearly specified. After this was done, a certain amount of experimentation was possible to get the best design of interface, and the most efficient application.

### The Maze Protocol

The construction of the protocol for the 3D maze game was substantial task. The first part was an investigation of the topology of the mazes it is trying to describe. There is naturally a large difference between a three dimensional representation of a flat two-dimensional maze, and the three dimensional representation of a truly three-dimensional maze. The protocol designer needs to take such factors into account.

The maze can be visualised as a cube made up of smaller cubes. Where the cubes join there is either a wall, or no wall. There is the guarantee that all locations can be reached by navigating the passages created by the missing walls.

So, in order to visualise this maze for the user, the interface needs to know which walls are absent from the user's point of view. This is achieved by the application core telling the interface which walls in front of the game player are there, and which are absent — or in simpler terms, which walls are "on" and which are "off". You could imagine a textual interface describing what paths are available from the user's position, given this information.

The complete protocol specification written by the interface designer and shared with the core program author is presented in Figure 3.5. In it the protocol designer's comment can be seen, that, of the walls potentially visible (twenty-one), only nine will be implemented initially. In a character interface perhaps fewer walls would be implemented or, in a more sophisticed graphical version, perhaps more.

As well as the maze display there is a 'status' line displayed. This is used to relay information from the application to the player, reporting the player's progress.

So there are a total of three commands which an interface has to provide to the core.

- `on` *i j*

  The `on` command simply turns 'on' a wall at distance *i* away from the user, wall number *j*, so that the player may not move there.

- `off` *i j*

  The `off` command is the inverse of the `on` command, making passages available for the player to navigate.

*For each level i, there are 21 walls, but only the first 9 have been implemented, numbered as j, 0 to 8.*

*To turn a wall on, use "on i j"*

*To turn a wall off, use "off i j"*

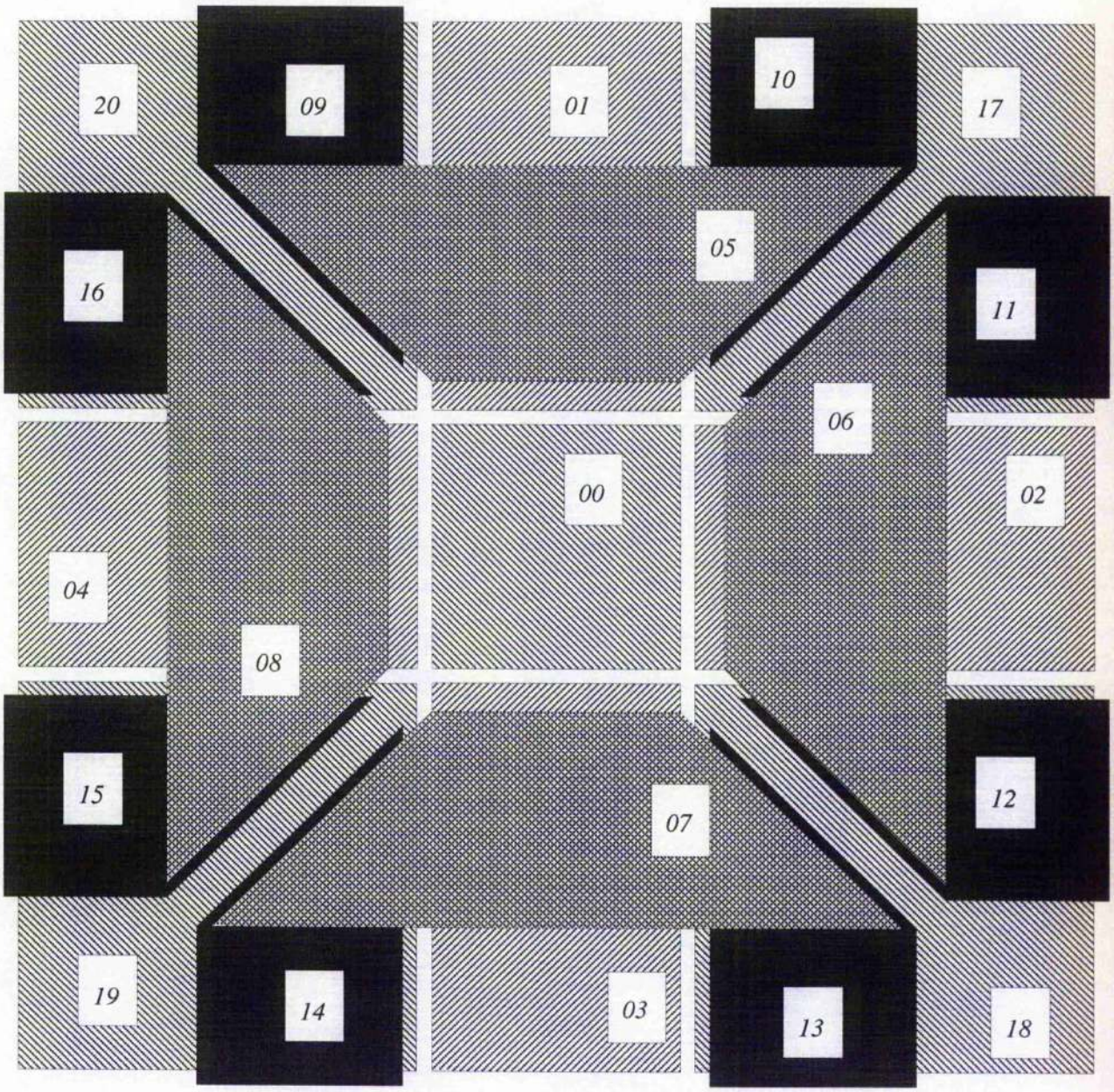


Figure 3.5: Specification of Wall Display

- `status` *c [i]*

  This informs the interface of the current status of play. The particular message is encoded into the character *c*, with any extra data required given in optional integer *i*. An example would be "status f 45" updates the message to indicate that the player has finished the maze, and that it took 45 moves.

It is important to notice that the actual 'status' message to be shown is generated by the interface, based on the code letter sent from the application. This allows easy adaptation of the program, for example when translating the program into a different language. In this case only the interface needs to be changed, and the application remains constant.

In the opposite direction, the most important user events are the movement commands. There are four possible moves understood by the application core, which the interface may wish to employ. These are 'move forward', 'turn right', 'turn left', and 'move backwards.' At all times, the interface can assume these commands are valid.

Two other user commands affect the application program; the user may restart the same maze from the beginning, or can choose to play on a new, different maze.

So there are six high-level events that the interface can send to the application.

- `in`
- `out`
- `left`
- `right`

  These four events signal player movement and will be triggered either by the player pressing buttons or typing keys.

- `init`

  Sets the player back to the start of the current maze.

- `new`

  Creates a new maze.

Once the protocol had been specified, the interface and application were then written completely separately.

### The Maze Interface

The maze is represented in the interface I have constructed by a three dimensional display of the walls, as seen from the game player's point of view. The walls have been created using the Tk canvas widget, and each wall has been drawn explicitly. Walls are turned on and off by altering the colours of these walls, making absent walls transparent. Directly below the walls is the status information.

To the right side of the main display is a panel of buttons to control movement. This area is the main source of user events.

The four events signaling player movement are triggered either by the player pressing on the directional buttons in the panel, or else using the keys 'h', 'j', 'k', and 'l' to control movement, in the Unix tradition.

In response to these events the application is expected to send a sequence of commands which will update the displayed scene in the maze, as explained above. There is, however, no knowledge within the interface of what a particular directional command means.

The two major sections of code in the interface are for building the interface and display. All other code is fairly trivial. Parts of the program which implement the interface can be seen in Appendix C.4.

### The Maze Application

The maze program is written as two Haskell modules. The most crucial is the module of functions which implement the maze, creating it, manipulating it and determining movement around it.

The maze is structured as a three dimensional array of boxes, where the size of each dimension is arbitrary. Each box has between 3 and 6 neighbours, with which it shares a wall. Paths are then cut through the maze by removing walls until all boxes are reachable from each other, creating a fully navigable maze.

The other module is almost exclusively concerned with the interface, translating user commands into maze operations and then passing back display information to the interface. This code is perhaps only a quarter of the complete program. Only a small part of this, perhaps one third, is involved with communication with the interface, the rest being the mechanics of the game, keeping track of where the player has been and

determining whether the maze has been completed yet or not.

This module might be seen as being a 'linkage' between the interface for the user, and the raw functionality of the maze manipulation code. The maze functionality is not written for the specific application and could be used in any exploration of a maze. It is this linkage portion which is the key to how the functionality is presented to the interface.

Interpreting the commands is fairly straightforward. The commands which turn the player around to face different directions are implemented by rotating the view in the maze about a point. The motion commands have to be handled specially. The program must ensure that the player is standing on a solid wall after the move. When faced with a solid wall directly in front of the player, the program allows the user to 'climb' the wall, in the way a spider might. When faced with a hole, the player is moved onto the side wall of the hole or, if that does not exist, the player will actually end up on the other side of the wall they had previously been standing on. Imagine the spider walking off the edge of a table to understand what happens.

The maze manipulation code is very rich in good functional programming techniques, using a great deal of composition of higher order functions, and applying mathematical theory for the construction of mazes. It may be studied further in Appendix C.5.

### 3.3.3 Summary

I have presented details of the workings of the interface for the two example programs. The dialogue that occurs between each interface and their application cores have been presented to show that I have achieved a high level of separation between application and interface in my programs.

## 3.4 Running Examples

In this section I present the examples from Chapter 2, showing how I have modified them and given them graphical user interfaces. The purpose behind this is to demonstrate what modifications are required to give an existing functional program a better user interface.

Both examples lack any real functionality, as they both exist as simple examples of interaction within functional languages. For this reason, the particular design and implementation of each application will not be considered a great deal in this investigation. To

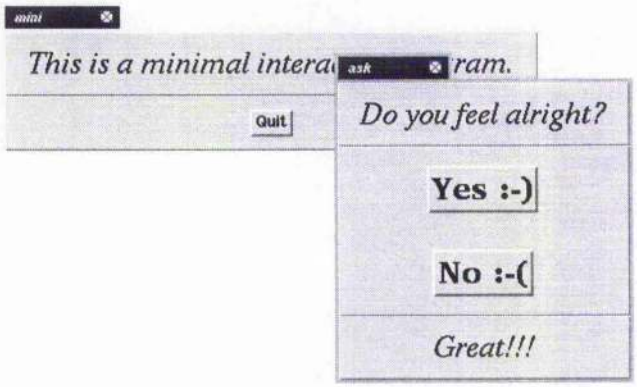


Figure 3.6: Minimally Interactive Program

get the best design would involve re-writing the program, which would not show how easy it was to adapt them.

### 3.4.1 Minimally Interactive Program

Figure 3.6 is a screen-shot of the graphical version of the 'minimally interactive program'. The user has just been asked "Do you feel alright?" and has answered by clicking on the 'Yes' button. The program's response is displayed below while the program waits for another response. At any point the user can exit from the program by pressing the 'Quit' button of the main window.

This program repeatedly asks how the user is. This is a natural "main loop", so no major changes were required in the structure of the code to make it suitable for graphical interaction. I used the continuations based version from Page 32 to build the new graphical version. The modified code is presented in Figure 3.7.

In fact the most obvious changes are three simple 'global substitutions', the first being a terminology change of 'input' for 'events', abbreviated as 'evs'. The other two are changes of the input channel from 'stdin' to 'epipe', the stream on which events arrive, and of the output channel from 'stdout' to 'cpipe', the command stream to the interface.

The other major change is that, instead of outputting questions directly to the user, we must encapsulate them into commands — the 'ask' and 'answer' commands.

The only addition to the code is definitions of 'cpipe' and 'epipe', as file descriptors which we are using as streams. A simplification to the code has also been made. As programmed, the interface will always give a lowercase 'y' response for 'yes', and so we

```
module Main(main) where

main       = getevents

getevents = readChan epipe exit                                    $
            \evs -> how (lines evs)

how evs    = appendChan
              cpipe "ask \"Do you feel alright?\"\n" exit          $
               case evs of
                (l:evs) -> case l of
                             'y':xs -> good evs
                             _      -> bad  evs
                _          -> done

good evs   = appendChan
              cpipe "answer \"Great!!!\"\n" exit                   $
                how evs

bad evs    = appendChan
              cpipe "answer \"Sorry to hear that.\"\n" exit        $
                how evs

cpipe = "/dev/fd13"
epipe = "/dev/fd14"
```

Figure 3.7: Minimally graphical interactive application program.

do not need to deal with uppercase at all.

The interface needs to be custom written for each new application. Portions of the interface code are presented in Figure 3.8, the full code is presented in Appendix C.6. It would make no sense to go through the code for the interface line by line, as this thesis is not a tutorial for Tcl/Tk, but the creation of the 'Yes' and 'No' buttons should be clear. Pressing either of these buttons will cause either an "event yes" or "event no" command to be executed, which will in turn send the appropriate event back to the application.

At the foot of the interface code is the call to the 'spawnchannels' command which I added to Tcl/Tk. This call has the effect of starting up the main application once the interface's main window has been created.

So, the changes to the actual functional program were very small and it retains its structure and general appearance. The resulting program is as readable as the textual version and adds no complexity for the functional programmer to understand.

I would argue further that the interface is relatively simple, requiring no great level

```
proc mainwindow {} {
  # code for main window creation removed

  button .bot.death -text "Quit" -command {destroy .}
  pack append .bot .bot.death {left expand padx 20 pady 20}
}

proc ask {text} {
  toplevel .ask

  # code for construction of query window removed...

  button .ask.mid.yes -text "Yes :-)" -command "event yes"
  button .ask.mid.no -text "No :-(" -command "event no"

  # further code removed...

}

proc answer {text} {
  .ask.bot.answer configure -text $text
}

# create main window...
mainwindow

# run main program...
spawnchannels how
```

Figure 3.8: Minimally graphical interactive interface.

of understanding over the basic commands which are used in Tk to build interfaces.

### 3.4.2 Bank Machine

Porting the ATM was quite a different task to the one before. The ATM was programmed to simulate the trace of interactions possible in a bank machine, but it was not intended as an example of a useful program. For this reason the structure of the program is the same as the interaction presented by the bank machine. In real life this structure would only occur in the interface and not the application.

It was thus necessary to build an interface which could interact with the very fixed modes of interaction in the application, rather than with a structure like an event loop or similar. The original version of the program does no checking of input, and will terminate with an error if it has trouble with user input. The same is true of the graphical interface,

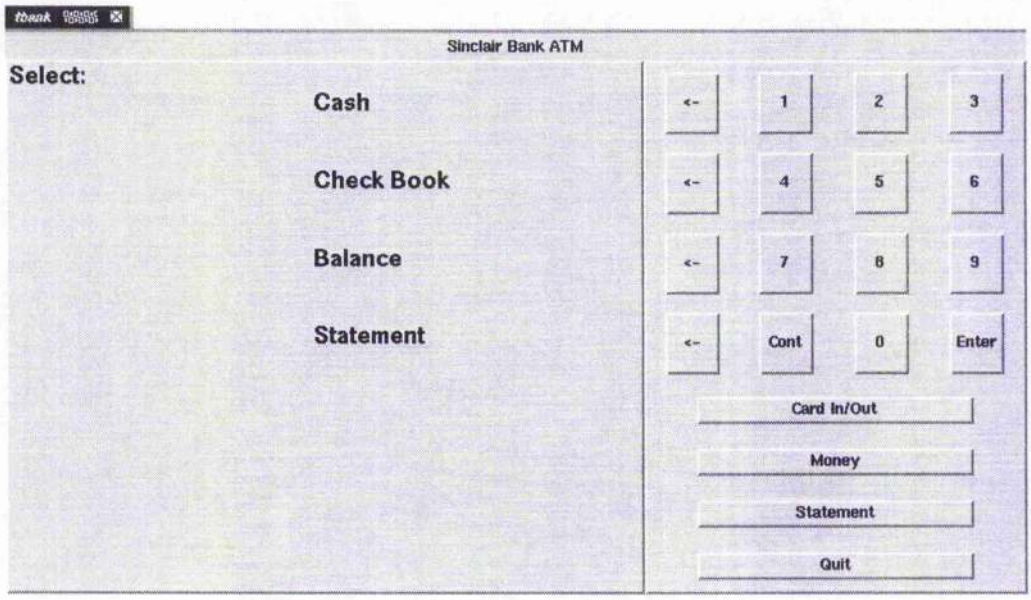


Figure 3.9: Interacting with a Bank Machine

which is merely layered on top of the textual interface. Figure 3.9 contains a screen-shot of the interface in action.

The interface was programmed to communicate with the application at approximately the same level as the user would in the textual version. Textual output from the application was encapsulated into commands for the interface, while user input was passed by the interface as entered.

All input to the program is numerical or simply to confirm an action, and so a simple numeric key-pad was provided on the display, with an "enter" button to feed numbers and confirmations into the program. For actions, such as taking money or entering a card, special buttons were provided, but these were, in effect, the same as the "enter" button. In addition, a column of buttons forming a menu beside the display was provided, but again, these buttons were simply short-cuts for numerical input.

Output was tricky for the simple reason that the textual version assumed that all messages written out to the screen could be read and that there was no limit to what could be displayed at one time. However, in the graphical version, we provided a small viewing screen which could only hold a small amount of text at a time.

This was solved by classing the messages into 4 different variations:

1. A message which could simply be written to the display.

2. A message which should be written to the screen after it is first cleared — that is, that nothing was written before it which the user has not had a chance to read.

3. A message which must be acknowledged before the program will continue.

4. A message which forms a menu, and makes use of the menu buttons placed alongside the display.

Again, the changes to the application are fairly limited. The code changes for the application are in Appendix C.8. Excepting the changes of the communication channels to use the pipes connecting to the interface, the only other changes were in the code which managed the messages output. This was changed to match the 4 different styles of output which we had decided upon above. Instead of all output messages being handled by the output function 'message', there is now also 'nmessage' which clears the screen first; 'ack' which presents a message and waits for a confirmation; and 'messages' which supplies a menu choice to the user.

The interface has been custom-written for the application. The complete code of the interface is given in Appendix C.7. The basic elements of the interface are an input panel with a numeric key-pad and other buttons, and the output area where messages are presented to the user and feedback from the numeric keypad is displayed.

The buttons are either wired to send an event string or, for the numeric key-pad, to add a digit to the number to send. The 'Enter' key then sends the stored number to the application.

The display area is managed by three commands, one of which clears the display, another displays a message, and the last one formats a menu for a choice to be made using the menu buttons.

So, we have a graphical version of a textual program, built with very few changes to the original program. The key part of the program — the description of the interaction of an ATM was completely unchanged. The changes required were mainly due to the problem that in the original ATM specification timings considerations were abstracted away.

The modifications to the program to give it a graphical interface took less than two hours, with the required modifications to the output functions adding around another hour.

### 3.4.3 Summary

I adapted two existing textual functional programs to give them graphical interfaces with Tcl/Tk. The changes to the program were minimal, with no major alterations required. The major tasks were designing the interface and the protocol of communication used between the interface and application.

## 3.5 Discussion

I wish to highlight a number of particular features in my design which add to its merits as a workable solution to user interface creation for functional languages, beyond what has been specified in the requirements from Chapter 1.

One important aspect is the simplicity of my solution. Its design and implementation were very straightforward, most effort going into ensuring that the resulting `swish` program is robust in the face of bugs in other programs. The evolutionary design of the system made the implementation easier, because I was already familiar with the operating system features I needed to use.

The evolution from previous solutions involved using a programmable system to manage the interfaces. I could have implemented such a system from scratch but, instead, minimised effort by using tools already in existence, and already proved able to do the job. If I had built my own programmable interface system, there would have been no guarantee that it would have worked and, if it had, it would have been a case of reinventing the wheel.

As a result of choosing to use Tcl/Tk, I have also gained the use of a more powerful interface creation system than if I were to build my own. Tcl/Tk was expertly designed, and this shows in the simplicity of programming notation required to build large, powerful interfaces. I am also able to use interface building tools to build interfaces, making it even easier to construct interfaces for functional programs.

It is important to repeat that all the interface construction, layout and management happens within the Tcl/Tk programs. This takes all this out of the scope of the functional program, where it can obstruct good programming techniques.

This means that there is more time and scope to use the features of functional languages, such as laziness and compositionality. The Haskell portion of the maze game is

especially rich in functional techniques.

Finally, as the interface is being written in the imperative languages Tcl, all the usual interaction with external systems not possible in a functional language can now be managed. For example, signals from operating systems can be programmed in standard ways. These can be handled within the Tcl/Tk system without affecting the operation of the functional program. If a signal requires some action on the part of the functional program, then it can be turned into a high-level event and handled in the normal manner.

## 3.6 Summary

I have presented a system whereby a functional program may be connected with a separate interface process, thus providing a means of user interaction for the functional program.

This external system is written using the Tcl/Tk system, a language and toolkit combination. Using Tcl the interface designer can create a fully functioning interface which may then be connected, by way of simple communication channels, to any other program which understands the communication protocol.

In my case, I was interested in providing this interface for functional programs, written in Haskell. Using the standard I/O mechanisms in the functional language, the interface and application program communicate, commands being sent to the interface in the Tcl language for direct interpretation by its interpreter. Events and replies coming from the interface are coded as simple strings, which may be parsed by the functional program.

The first two examples show how easy it is to build interfaces for functional programs in Tcl/Tk, and how easy it is to communicate with these interfaces using simple commands to instruct the interface.

The examples taken from Chapter 2 show how easy it is to adapt existing programs to put a graphical interface on top of its textual one.

In the next chapter I shall discuss to what extent my solution meets the requirements set out in the Introduction.

# Chapter 4

# Assessment

In this chapter I review my system to see if it meets the requirements as I have laid them out. I will also highlight any strengths or weaknesses in the system, which can lead to improvements in the overall system.

## 4.1 Requirements

I have set a number of requirements over the course of this thesis, initially set out in the introduction, then extended further in Chapter 2. I shall now review them quickly.

- Requirements from Functional Programming

  The programs written to use my system must be programmed in a functional manner and it is not acceptable to modify the language in a way that would compromise referential transparency.

- Software Engineering

  The system I create must be capable of creating good modular programs. Maintenance and portability of these programs are very important.

- HCI

  The process of interface creation is iterative, participatory and exploratory. I must ensure these elements of interface design are properly supported.

- UIST

  The first aspect of UIST requirements demands that the interfaces are separated from the application core. This helps make possible the demands of the HCI issues.

  The other key area is that of supporting multiple concurrent threads of operation. Within each thread a programmer may wish to impose some sequencing, but separate dialogue threads in different areas of the interface must be able to operate independently of each other, without interference.

- Overall Requirements

  There are requirements which all of the above areas demand.

  The first is that of effort, both in design and construction of the program and its interface. There is no point in creating a system which meets all the other requirements if it is impossible to design and implement useful programs with it. The system I produce must be easy to use.

  Portability is always a desirable element. I should avoid making choices which lock programmers into one environment.

  Lastly, I should plan for the future, and allow the system I build to grow with people's requirements. I should also be aware that a flexible approach is required to allow for new developments. Whatever system I devise should have good extensibility.

- Further Requirements

  After my investigation of Functional I/O in Chapter 2, I enhanced my requirements with some specific points which would apply to the functional programs that would be written.

  I need to ensure modularity within the functional program. The code which manages the interface cannot get entangled with the actual application code.

  I found that some I/O solutions were not immediately extensible to allow for future flexibility. I should ensure this does not happen.

  It is important that the programming effort within the functional program of I/O is not too high. It is important that a simple I/O system is used.

I wish to support a multi-threaded style of functional programming, which would match the multi-threaded nature of interaction.

## 4.2 Do I meet the Requirements?

I believe that my system meets all the requirements outlined above. While the additional functional programming requirements are not perfectly matched, I feel that there is nothing inherent in my design which would prevent further work in this area to move it nearer the actual requirements. In all other areas, I believe that I meet or exceed the requirements.

For each of the requirements, I shall now discuss below whether my solution matches them and, if a particularly good match, any additional benefits accrued from my system.

### 4.2.1 Requirements from Functional Programming

It is important that programmers are allowed to use the particular features of functional languages when they write their programs and are not forced to compromise their design to fit in with the interaction style. I believe that I accomplished this.

By programming the interface in a separate system, the only interaction that the functional program would have to take part in would be at a very high level with the interface. This allows the functional programmer to concentrate his programming efforts on the main task and not have to worry about the complexities of interacting with users. I argued in Chapter 2 that functional languages were not well suited to programming user interaction.

My second example program, the maze game, uses functional features, such as composition and laziness, a great deal. When handling the interaction, continuation-style I/O was employed which is very compositional and is well suited to small amounts of interaction.

I also stated that I should not compromise referential transparency within the functional framework but, as I have not needed to adapt the functional language at all, I easily meet this requirement.

### 4.2.2 Software Engineering

Programs written using my system must be modular. Maintainability and portability are very important. Unfortunately, these concepts are very hard to measure.

My example interfaces were too small to get a good idea of how modular their code was, but I have a clear distinction between code which is used to create the interface, the code which is used to maintain it, and the code used to communicate between interface and application. Their maintainability can only be guessed at, but their typically small size must help here. Portability rests on Tk/Tcl and I have avoided using any particular system-dependent features.

Inside the application code, I have separated code which deals with the interface and interaction from code which deals with application data, and its manipulation. Because of this, maintainability is kept high and portability is limited only by the way in which the functional I/O system interacts with the operating system the program is running upon.

### 4.2.3 HCI

Tcl, as an interpreted language, cuts out the compilation phase, leading to faster turn-around of interface design. This leads to a fast loop in an iterative design loop.

Much more important is that Tk will allow external processes to communicate with the running program; for example, to up-load revised versions of procedures or to change the values of variables. This allows the interface to be created interactively, textually or using a combination of the two techniques. It is possible to actually adapt the interface while a user is working with it, allowing high levels of participation in the design of the interface and making exploration easy and fast.

### 4.2.4 UIST

The key area which needs to be addressed from UIST is separation. I look at this first, discussing other aspects of UIST afterwards.

#### Separation

My complete system was built upon the concept of separation, so it is no surprise that I do particularly well here. I have complete divorce of control between the interface and

the application program.

The discussion in Section 2.6.1 explained how it is easy to break software engineering guidelines by unnecessary coupling between user interface code and application code. I want such interaction to be minimised in my implementation. My separate interface provides this.

By operating the interface of a program as a separate process it is more difficult to compromise the modularity of the code by having too much coupling. The interface communicates with the application, but as separately written bodies of code, and so the coupling is minimised. An especially useful consequence of this is that it becomes much easier to modify the interface without requiring much, if any, restructuring of the rest of the program.

This does not go as far as the complete separation put forward by some HCI researchers[6] where it would be possible to completely change the structure and style of the interface without modifying the application — separation of representation. However, taking the initial small step of having the interface constructed separately is a sufficient goal for which to aim. For complete separation a greater abstraction would be required between the interface and the application program. Currently, the application needs to have some knowledge of aspects of the interface and, likewise, the interface needs to know things about the structure of the application, such as requiring that the application works in an event style of programming. A third component of the system could manage communication between the interface and application, coordinating their interaction and removing assumptions they have to make about each other.

I have demonstrated that the functional application can be developed separately from the interface, with the example of the maze game which was developed in two different countries.

### Other aspects of UIST

Tcl provides a sequencing within its language. Concurrent interaction is achieved by running the interface as a separate process from the application, so user input will continue to be handled while the application is busy.

I have not addressed threading directly, but some amount of threading is natural in

Tcl/Tk programs. Within the functional language, threading is not done.

An important area I have also failed to address fully is error-handling, both in terms of system errors and the user's mistakes. This should be addressed in any further explorations on the interface side of this work.

### 4.2.5 Overall Requirements

These were the requirements common to all the above areas.

#### Ease of Design

Designing interfaces in Tcl/Tk is easy. First there exist tools which make it easy, allowing interaction lay-out of an interface, with simple programmatic tasks being written for you.

Designing the communication protocol between the interface and application is a matter of deciding the semantics and functionality available in the interface. It should be possible to express all this in a concise format within the protocol.

As for the design of the application, it is easier than before, where all the interface had to be included in the design or the program, along with all the functionality.

By separating the design into these three areas, it becomes easier to modularise the design phase and the difference between interface and functionality becomes clearer.

#### Ease of Construction

The Tk toolkit is very easy to use, allowing people with no experience of programming for the X Window System to create simple programs after only hours of experience with Tcl and Tk. Of course, to get the most out of Tk requires careful study, but remarkable complexity of design can be achieved with relative ease.

The functional programmer's interface to the user interface, being via the standard I/O system of the functional language, is no worse than any other current way of programming user interfaces from a functional language. Any user interface system implemented within a functional language that does not communicate via the I/O system could be used to control a Tk interface. Any new abstraction over the I/O system could also be employed to communicate with the interface.

I argue that, just as Tcl/Tk makes creating graphical interfaces for programs written in C easier than just using C on its own, the same benefits are to be found by using Tcl/Tk rather than a functional language for interface creation. I feel it is important to acknowledge that special-purpose languages can produce better results, easier than general purpose languages, either imperative or functional.

#### Portability

Tcl creates a slight layer of abstraction over the normal operating system functions, allowing the same Tcl program to run unchanged on different versions of the same (Unix) operating systems and with little change between different operating systems.

Likewise, Tk abstracts from features of the X Window System, making it possible to move between versions of X and different displays with different features, without requiring special handling within the program. In the future it is expected that versions of Tk will exist which will run on Macintosh and Microsoft Windows, allowing easy porting, i.e. with very little modification required, of Tk/Tcl programs between very different operating systems.

By handling all these issues outside of the functional language and programs, porting the functional language between different machines is made easier. Also, because the functional programs do not use an embedded interface to a window system in my system, no modifications are needed if extensions of the system are required. If I used an extension to the language, then incompatibilities could be introduced when the devised interface does not abstract sufficiently from the implementation.

#### Flexibility

My system gains all its interface flexibility from Tk. Tk has been used to create many diverse programs, from simple games to complex presentation creation systems. Plus, if Tk is found to be lacking in any particular feature, then it is easily extended: many extensions already exist for Tk, proving how simple this is.

Often Tcl and Tk are used to create graphical interfaces for programs which are not interactive or not so sophisticated in their interaction. This is basically what my system does, except that the unsophisticated interactive programs in this case have been written

in a functional language and a discipline of communication has been specified.

As I am not changing the functional language, I do not affect its inherent flexibility. It must be pointed out, however, that unlike other systems, I do not put any restrictions on the I/O system in use in the functional language.

### 4.2.6 Further Requirements

I had some further requirements specifically from the point of view of the functional program, and the use of its I/O system. Some of these points are simply lending extra weight to the requirements already given. The main emphasis, however, is that I do not restrict the way functional programmers goes about their task.

#### Modularity of FP

With respect to modularity of the functional programs, my system does not impose a particular style of functional programming that inhibits intrinsic modularity within a program. This can be seen as meeting the requirement.

At the same time, the application has an overall benefit by having the interaction code removed into a separate system. The removal of interface code will make the functional code cleaner in design, with less management of interaction, which can be troublesome in functional languages.

#### Flexibility in I/O system

It is important that the way users communicate with the functional programs through the interfaces is not restricted to current ideas, but that the environment can grow to meet future requirements. I have been using the continuations I/O system to communicate with the interface. As this communication is simple text, there is no danger of unknown features being unavailable due to lack of power in the functional I/O system. However, there is the possibility that more structured communication is required some time in the future.

#### Effort

By taking the handling of the interface out of the functional program, I have simplified the programming task. The functional programmer still has to deal with high-level events, but these are by nature well specified and do not need the careful handling that low-level events require in order to achieve good interaction.

The communication between the interface and application core is purely textual. This is very easy for the functional programmer to deal with, splitting the input up into lines, which can be easily pattern matched. Output is, again, line-based and is made very easy for the functional programmer using the standard I/O system of the language.

#### Multiple threads in FP

Multi-threaded execution of the functional program is an area I have not addressed at all. I have taken an event-loop structure for my functional programs and this, to an extent, gives an illusion of multi-threading, but multiple threads of state are what is missing, and so my programs are still fixed to a sequential evaluation model, with the programming overhead of current state being passed around all parts of the program. This is an area of current research, and I will come back to it in the conclusions in Chapter 5.

### 4.2.7 Summary

With its clean interpreted style, Tcl makes a good language with which to build user interfaces. The Tk toolkit built on top of Tcl provides a complete system for creating interactive programs. Its ability to multiplex multiple input and output streams allows it to build responsive interfaces which can interact with the user and application at the same time.

## 4.3 Strengths and Weaknesses

There are places where my system does not match up with the ideal. There are also places where my system excels, simply because of some of the decisions made in its design.

Perhaps the most obvious flaw is that functional programmers, in order to create interfaces using Tk and Tcl, need to learn the imperative Tcl language, which can surely

not be as powerful as the functional languages they are used to. The answer to this is simple: user interface creation is not the same as programming. Interface creation is becoming a more specialised job, not involving as much programming but, instead, it involves tools tailored for the creation of user interfaces. Many of these tools require the use of their own language for specifying aspects of interaction which are not necessarily programmatic in nature. Tcl has full programmability, making it more powerful than many other languages used in such applications, while its use is normally limited to quite a simple subset of its facilities. At a more pragmatic level, learning to use the Tk toolkit will be no harder than any other way of communicating with a window system to implement an interface.

I have a very strong reliance on Tk and Tcl. As a result, I am limited to what they offer, although I could resort to programming, in a different language, to enhance Tk and Tcl, incorporating any features I might need. For existing applications, few other authors have needed to extend either Tk or Tcl, although some require one or more of the readily available extensions, which are also available to the functional programmer, if needed. Often the key reason why people are forced to program extensions to Tcl/Tk is to speed up the application's processing. As the applications are already in a fast compiled language, this should not be a concern.

I run the interface as a separate process from the functional program. Some operating systems are not able to do this and so I have limited the ability to port my system. However, any operating system, with some form of multi-threading, will be able to use the same basic techniques. Without some form of concurrency, any system which provides graphical interfaces to functional languages would have to be careful about lack of response from the interface when the functional program is executing. It is for this reason that I have used separate processes, and so I avoid this problem.

My biggest strength is the simplicity of Tk. Tk is far simpler to learn to program than the raw programmer's interface to the window system. It is also much simpler than most toolkits. Tk interfaces are easy to write. Tk sits at a relatively high level of abstraction, and Tcl creates such a clean programming environment that Tk programs can be a tenth of the size of competing systems. The ubiquitous "Hello World" program in standard OSF/Motif is 38 lines long, while in Tk/Tcl it is only 2 lines long.

Despite my current reliance on Unix discussed above, my approach, although per-

haps not my particular implementation, is highly portable and could allow a functional program to be moved between different machines with only a recompilation. To port the interface, assuming that Tk exists on the target system, would require little work. Without having Tk on the remote system, as long as some similar system exists, such as Visual Basic on Microsoft Windows, it would be possible to re-write the interface in that language, without needing to re-work the functional code.

My biggest feature is that the interface is created and exists separately from the application. This allows rapid prototyping and testing of interfaces while functional code is incomplete. Tk is an ideal tool for rapid prototyping. The interface creator can directly work with the interface, while it is running, using the Tcl language, both testing the application's programmatic interface, and modifying and customising the user interface directly. Using a user interface creation tool allows the programmer to test-drive the interface, and to modify it seemlessly.

Separation of interface allows programs to be developed separately, the application code being written by one person, the interface by another. Once there is a high-level protocol defined between the application and its interface, the two programmers can work totally independently, only bringing the two parts together when complete.

Tk with Tcl is powerful. The interfaces created using Tk do not lack features compared to other systems which might appear better due to their greater complexity. While the Motif toolkit has features that Tk lacks, the reverse is also true.

## 4.4 Summary

I believe that my system of using Tcl/Tk to create user interfaces meets the requirements laid out. I have found an especially useful facility in Tcl/Tk, to create interfaces, simplifying the job of the functional programmer, who is saved the trouble of complex programming of user interfaces in a functional language.

# Chapter 5

# Conclusions

I conclude with summaries of the background to my work and what I have done, followed by a list of my achievements and how my work could be further exploited. I then set out a number of areas where this research could be extended.

## 5.1 Summary of Background

I set out to tackle the problem of creating good user interfaces for programs written in functional languages. I shall review why this is an interesting problem. Firstly, why bother with functional languages?

Functional languages take a very high level approach to programming, where they describe mathematically the solution to a problem, thus implying a computational method, rather than explicitly specifying what operations are required, as is needed in the more traditional imperative languages. This gives the programmer a much more expressive language to work with, making shorter, more powerful programs.

During the time functional languages have been developed, user interaction techniques have developed, allowing users to interact with programs in a simple and easy fashion, typically through a window-based interface, controlled by a mouse. These interfaces have brought more power to the user by providing easy ways to do complex things.

Unfortunately, programming graphical user interfaces has always been done in a very imperative style, reflecting current techniques. Until recently, little work had been

done to adapt either functional languages or user interface toolkits, to allow them to work together. Functional languages have not been able to exploit current technology from Human-Computer Interaction (HCI) and User Interface Software Technology (UIST) research.

By adopting principles from other branches of computer science, I set out to find a solution to this problem. I took with me principles of program design from software engineering, rules of interface creation from HCI and, finally, techniques for programming user interfaces from UIST.

## 5.2 Summary of Work

I have surveyed existing methods of simple textual I/O in the language Haskell, as a representative of functional programming languages. It is currently rich in I/O techniques, with the well-tried traditional dialogue style; its cousin, continuations; and the new technique, monadic I/O. I concluded that continuations were much easier to use for simple tasks than the other two, but monadic I/O wins out in the end due to its greater over-all flexibility.

Moving on from textual I/O, I examined two systems which allow user interfaces to be created from within a functional program. The first, Fudgets, is a totally functional solution, developed on top of the existing dialogue I/O system from Haskell, with some simple extensions to allow the language to communicate at a low-level with the window system. The second was built into the language Concurrent Clean, which provided primitives and a novel type system to allow the programmer to invoke user interface functions in a functionally pure style. Both these approaches I found to be awkward, requiring a difficult programming style which is alien in the clean world of functional languages. Neither allows programmers to exploit user interfaces designed by UIST tools.

With all this behind me, I set out to create a powerful system for creating user interfaces for functional languages, while retaining the purity and style of the functional language. I did this by creating the user interface outside of the functional world, but tied the interface to the functional program through a high-level dialogue, which the functional program would interact in using conventional I/O methods. This was a key point from UIST, which showed that interfaces should be highly separated from their application

programs.

The interfaces for my programs are created using a language called Tcl, a simple interpreted imperative language, and the user interface toolkit, Tk. Together, they allow highly complex interfaces to be built with remarkable ease. By using the Tcl language to allow the interface to be controlled by the functional program, I was also able to give a very high level of interaction between program and interface. The Tcl/Tk system, by its nature, allowed a highly flexible interface development style, as guided by HCI research.

I implemented this by extending the Tcl language to allow it to spawn a new process, a functional program, with which it can communicate via three channels. These are used by the functional program to communicate with, and control, the interface. Because the interface existed as a separate process to the application, many of my requirements, involving modularity and interactive response, were easily met.

Experiments with the system involved creating two sample applications — a very simple clock, and a more complex three-dimensional maze game. I found it to be easy to create interfaces to the functional applications which had been written. The maze application program and the interface were written separately by two people, showing that the interface was created through a separate design process from the application.

Further trials involved taking the examples developed in Chapter 2, and giving them new interfaces using my system. This shows how my system is a relatively uncomplicated addition on top of the existing I/O system of a functional language.

The four example programs show my system to be workable, meeting all my early requirements in full. Particular requirements concerning how the functional program should be written, and how I/O within functional programming might be improved, were not examined especially closely, as they would inevitably require development and modification to the basic language, but I believe that I have made some improvement by removing interaction code from the functional program, where it obscures the clarity of the functional computation.

## 5.3 Achievements and Possible Developments

My achievements are as follows:

- I have developed a system for building graphical interfaces to functional programs. This uses an external program to provide the interface, which communicates with the functional program using a high level of abstraction. This allows the functional programmer to devise functional programs with less worry of how it will interact with the user.

- I have enabled principles of program and interface design to be applied to the construction of interfaces for functional programs.

  From software engineering, I have used the concept of cohesion and coupling to guide programmers in producing modular programs.

  From HCI, I have applied principles of interface design to guide my choice of interface creation system, ensuring that interfaces for functional programs are easy to design.

  From UIST, I have employed guidelines which mean that the interfaces created with my system are flexible and usable, while being easy to program.

- Technically, I have extended the Tcl/Tk system, and so created a means of building communication links from the Tcl/Tk system to programs written in other languages. The modifications required to the run-time system of the Haskell B compiler system I was using were minimal, and were subsequently adapted and adopted by the compiler's author.

- I have shown that this system is usable in both small and larger applications.

  I created a small interactive alarm clock program which, by necessity, has a periodically updated display. This display remains active no matter what other interactions are also happening, showing that programs which must respond attentively to the user are possible.

  A larger program, a maze game showed that more complex interaction is possible, where large graphical displays could be managed by my interfaces, controlled by a functional program.

- I have shown how existing programs can be adapted to give them a graphical interface, rather that a textual one. This involved taking the examples developed

in the discussion of I/O systems for functional languages and adapting them to my system.

It is possible for others to apply this work in further ways. A number of people have experimented using the same system to allow better interfaces to be created for non-functional languages which also have suffered problems with interaction. It would be interesting to see how suited the system is to large-scale applications, involving multiple windows and a greater level of interaction.

## 5.4 Further Work

Further investigation should be given to ways of structuring the development of functional programs, to find a natural way to codify such things as event loops or callbacks or to find better abstractions which are more suited to the functional style. This would allow functional languages to be structured in ways better suited for interaction.

Multi-threading should be investigated in the context of functional languages. Currently, laziness gives a natural form of multi-threading based upon data demand, but I would like to investigate ways of running co-operating threads of execution which do not, or rarely need to, communicate. Chapter 2 referred to a scheme which would allow multiple output threads, involving the creation of new demand-driven output channels. I believe this to be a poor choice, for the same reasons that I believe the lazy inputs channels, as currently implemented by Haskell, to be a poor feature. Instead, if the idea were to be extended, to have multiple I/O worlds which could rendezvous to exchange data, this might be one way of introducing threading.

In this work I have only considered interaction with users. This can be seen as a specialisation of other types of interaction, such as the interaction a program would have with an operating system. For portability reasons, it does not make sense to define specific operating system interfaces in a functional language. This would result in reduced portability to different operating systems which might not support the same feature set, or could require a different style of interaction to achieve equal results. Instead, for the same reason as I prefer to deal with user interaction outside of the functional language, I would like to take all operating system interaction out of the functional language, and into a system like Tcl/Tk. In fact, there are extensions to Tcl which allow for large amounts

of system interaction which could be pursued further.

HCI research goes very much further than I have along the road of separation between user interface and application core. Cockton [7], for example, separates the complete program into interface, application and, between them, a 'linkage'. It is the linkage component that maps between what the application expects of the interface, and the interface expects of the application. The linkage is allowed to have knowledge about interface and application, permitting them in turn to be totally ignorant about each other. This gives even higher portability of interfaces and application, which can be created using specialist tools which would not need the extra weight of requiring tailoring to a particular mode of interaction. I would like to investigate what effects this would have on ease of creating interactive functional programs.

# Bibliography


[1] PM Achten, JHG van Groningen & MJ 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.

[2] PM Achten & MJ Plasmeijer, "The Beauty and the Beast," TR 93-03, Dept of Informatics, University of Nijmegen, March 1993.

[3] Lennart Augustsson, "A compiler for lazy ML," in *Proc ACM Symposium on Lisp and Functional Programming, Austin, Texas*, Aug 1984, pp. 218–227.

[4] M Barr & C Wells, *Category Theory for Computing Science*, Prentice Hall, 1990.

[5] Magnus Carlsson & Thomas Hallgren, "Fudgets – Graphical User Interfaces and I/O in Lazy Functional Languages," Chalmers University, Sweden, May 1993.

[6] G Cockton, "A New Model for Separable Interactive Systems," in *Human-Computer Interaction — INTERACT '87*, H-J Bullinger and B Shackel, ed., North-Holland, Amsterdam, 1987, pp. 1033–1038.

[7] G Cockton, "Architecture and Abstraction in Interactive Systems," Heriot-Watt University, PhD Thesis, Edinburgh, 1993.

[8] Andrew Dwelly, "Graphical user interfaces and dialogue combinators," ECRC, 1989.

[9] Ernest Edmonds, "The Separable User Interface," Academic Press, separation collection survey, 1992.

[10] MCJD van Eekelen, HS Huitema, EGJMH Nocker, MJ Plasmeijer & JEW Smetsers, "Concurrent Clean Language Manual — version 0.8," TR 92-18, Dept of Informatics, University of Nijmegen, 1992.

[11] Andrew Gordon, "Functional Programming and Input/Output," PhD Thesis, University of Cambridge, 1992.

[12] C Hall, K Hammond, W Partain, SL Peyton Jones & PL Wadler, "The Glasgow Haskell Compiler: A Retrospective," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland, 1992.

[13] CAR Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[14] Ian Holyer & David Carter, "Concurrency in a Purely Declarative Style," in *Functional Programming, Glasgow 1993*, Ayr, Scotland.

[15] P Hudak & et al, "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27 (May 1992).

[16] P Hudak & RS Sundaresh, "On the expressiveness of purely-functional I/O systems," YALEU/DCS/RR-665, Department of Computing Science, Yale University, March 1989.

[17] RJM Hughes, "Why functional programming matters," *The Computer Journal* 32 (Apr 1989), 98–107.

[18] David King & Philip Wadler, "Combining Monads," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Springer-Verlag, Workshops in Computing, Ayr, Scotland, 1992.

[19] R Milner, "A theory of type polymorphism in programming," *JCSS* 13 (Dec 1978).

[20] Gustaf Neumann & Stefan Nusser, "Wafe: An Interface to Xt and Athena," Part of wafe distribution, May 1992.

[21] JT O'Donnell, "Dialogues: a basis for constructing programming environments," in *Proc ACM Symposium on Language Issues and Programming Environments*, Seattle, Jan 1985, pp. 19–27.

[22] Open Software Foundation, *OSF/Motif Series (5 volumes)*, Prentice Hall, 1990.

[23] John K Ousterhout, "Tcl: An Embeddable Command Language," in *Proc. USENIX Winter Conference 1990*.

[24] John K Ousterhout, "An X11 Toolkit Based on the Tcl Language," in *Proc. USENIX Winter Conference 1991*.

[25] John K Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, April 1994.

[26] N Perry, "The implementation of practical functional programming languages," PhD thesis, Imperial College, London, 1991.

[27] SL Peyton Jones & PL Wadler, "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages*, ACM, Jan 1993.

[28] GE Pfaff, *User Interface Management Systems*, Springer-Verlag, 1985.

[29] Robert W Scheifler & Jim Gettys, "The X Window System," *ACM Transactions on Graphics vol. 5, No. 2* (Apr 1986).

[30] Duncan C Sinclair, "Solid Modelling in Haskell," in *Functional Programming, Glasgow 1990*, Workshops in Computing, Springer-Verlag, Aug 1990, pp. 246–263.

[31] Duncan C Sinclair, "Lazy Wafe : Graphical Interfaces for Functional Programs," Draft, University of Glasgow, July 1992.

[32] Duncan C Sinclair, "Graphical User Interfaces from Functional Languages," Final Year Project, May 1989.

[33] Satnam Singh, "Using XView / X11 from Miranda," in *Functional Programming, Glasgow 1991*, Workshops in Computing, Springer-Verlag, Aug 1991.

[34] William Stoye, "Input and Output," in *The Implementation of Functional Languages Using Custom Hardware*, December 1985, pp. 8.1–8.12.

[35] Sun Microsystems Inc., "LightWeight Processes," in *Programming Utilities and Libraries*, March 1990, pp. 2.1–2.43.

[36] Roger Took, "Surface Interaction: A paradigm and model for separating application and interface," in *CHI '90 Proceedings*, April 1990, pp. 35–42.

[37] PL Wadler, "The essence of functional programming," in *Proc Principles of Programming Languages*, ACM, Jan 1992.

[38] PL Wadler, "Comprehending monads," in *Proc ACM Conference on Lisp and Functional Programming, Nice*, ACM, June 1990.

# Appendix A

# Example from Introduction

## A.1 Graphical Interface Hello World in C

```
/*
 * xhello.c - simple program to put up a banner on the display
 */

/* Header files required for all Toolkit programs */
#include <X11/Intrinsic.h>      /* Intrinsics definitions */
#include <Xm/Xm.h>              /* Standard Motif definitions */

/* Public header file for widgets actually used in this file.  */
#include <Xm/Label.h>           /* Motif Label Widget */

main(argc, argv)
int argc;
char **argv;
{
        XtAppContext app_context;
        Widget topLevel, hello;

        topLevel = XtVaAppInitialize(
                &app_context,           /* Application context */
                "XHello",               /* Application class */
                NULL, 0,                /* command line option list */
                &argc, argv,            /* command line args */
                NULL,                   /* for missing app-defaults file */
                NULL);                  /* terminate varargs list */

        hello = XtVaCreateManagedWidget(
                "hello",                /* arbitrary widget name */
                xmLabelWidgetClass,     /* widget class from Label.h */
                topLevel,               /* parent widget */
                NULL);                  /* terminate varargs list */

        /*  Create windows for widgets and map them.  */
        XtRealizeWidget(topLevel);

        /*  Loop for events.  */
        XtAppMainLoop(app_context);
}
```

# Appendix B

# Examples from Chapter 2

## B.1 Continuation-style ATM

```
module Main where

{-
--      The "bank" program...  A simple auto-teller.
--
--      We're going to use continuations for this one.
--
--
-}

main    =       readChan stdin exit     $
                \input  ->      atm (lines input)

--      The behaviour of an atm specified in continuations...

atm     ::      Result
atm     =       insert_card_message     $
                card_in                 $
                enter_pin_message       $
                customer

customer        ::      Result
customer        =
                pin_no  $
                \pin    ->      validate_pin_no pin     $
                \valid_pin      ->      case valid_pin of
                        Pin_OK  ->      service_prompt          $
                                        services
                        Retry   ->      retry_message           $
                                        customer
                        Thief   ->      keep_card_message       $
                                        atm
                        Wally   ->      learn_number_message    $
                                        eject_card              $
                                        atm
                        _       ->      mydone

services        ::      Result
services        =
                service $
```

```
                \which  ->      case which of
                        Request_Cash            ->      cash
                        Request_Check_Book      ->      ack_check_book  $
                                                        more
                        Request_Balance         ->      show_balance    $
                                                        more
                        Request_Statement       ->      do_statement    $
                                                        more
                        _                       ->      mydone

cash    ::      Result
cash    =       amount_prompt   $
                amount          $
                \amount_query   ->      case amount_query of
                        Amount_Hopeful  ->      sorry_but_message       $
                                                cash
                        Amount_OK       ->      confirm_prompt          $
                                                confirm $
                                \doit   ->      case doit of
                                        Confirm ->      proffer_card    $
                                                        take_card       $
                                                        proffer_cash    $
                                                        take_cash       $
                                                        atm
                                        Cancel  ->      more
                                                ->      mydone
                        _               ->      mydone

more    ::      Result
more    =       eject_card              $
                another_service_message $
                eject                   $
                \answer ->      case answer of
                        Continue        ->      service_prompt  $
                                                services
                        Card_Out        ->      atm
                        _               ->      mydone

---

--      messages...

insert_card_message     =
                message "Please insert your card for service..."
enter_pin_message       =
                message "Please type your PIN."
keep_card_message       =
                message "Sorry, too many tries, I'm keeping it!"
learn_number_message    =
                message "Sorry, wrong number!"
retry_message           =
                message "Incorrect PIN, please try again."

service_prompt          =
                message "Please select a service 1-4."
ack_check_book          =
                message "A cheque-book will be sent out to you."
show_balance            =
                message "Your balance is <some-amount>"
do_statement            =
                message "Please take your statement."

amount_prompt           =
                message "Please type an amount of cash."
sorry_but_message       =
                message "You'll be lucky!"
confirm_prompt          =
```

```
                message "1 to continue, 2 to cancel"
proffer_card    =
                message "Please take your card."
proffer_cash    =
                message "Please take your money, have a nice day!"
eject_card      =
                message "Here's your card."
another_service_message =
                message "take it with 1, put it in again with 2."

--

--      hit return...

card_in         =       hit_return
take_card       =       hit_return
take_cash       =       hit_return

--      functions for the "Reply" types...

data    Valid_Replys    =       Pin_OK | Retry | Thief | Wally
data    Service_Replys  =       Request_Cash
                        |       Request_Check_Book
                        |       Request_Balance
                        |       Request_Statement
data    Amount_Replys   =       Amount_Hopeful | Amount_OK
data    Confirm_Replys  =       Confirm | Cancel
data    Eject_Replys    =       Continue | Card_Out

int2valid       1       =       Pin_OK
int2valid       2       =       Retry
int2valid       3       =       Thief
int2valid       4       =       Wally

int2service     1       =       Request_Cash
int2service     2       =       Request_Check_Book
int2service     3       =       Request_Balance
int2service     4       =       Request_Statement

int2amount      n       | n < 30        =       Amount_OK
                        |otherwise      =       Amount_Hopeful

int2confirm     1       =       Confirm
int2confirm     2       =       Cancel

int2eject       2       =       Continue
int2eject       1       =       Card_Out

--

--      functions to take user input into various types...

validate_pin_no ::      Int -> (Valid_Replys -> Result) -> Result
validate_pin_no =       continuation int2valid

pin_no  =       number_input id

service =       number_input int2service

amount  =       number_input int2amount

confirm =       number_input int2confirm

eject   =       number_input int2eject


continuation    ::      (b -> a) -> b -> (a -> Result) -> Result
```

```
continuation    x y z   = z (x y)

noop    x       =       x

--      difficult stuff...

type    Result  =       [String] -> [Response] -> [Request]

type    NumCont =       Int -> Result

message ::      String -> Result -> Result
message mess xx =
        \input -> appendChan stdout ("\n"++mess++"\n") exit (xx input)

mydone  input   =       done

hit_return      ::      Result -> Result
hit_return      cont    =
                \input  ->      case input of
                        (l:ls)  ->      cont ls
                        []      ->      done


number_input    ::      (Int -> a) -> (a -> Result) -> Result
number_input    f cont  =
                \input  ->      case input of
                        (l:ls)  ->      cont (f (read l)) ls
                        []      ->      done

--
```

## B.2 Monadic I/O ATM

```
module Main (mainIO) where

import PreludeGlaIO

{-
-- The "bank" program...  A simple auto-teller.
--
-- This time using Monads.
--
-}

mainIO :: IO ()
mainIO = readChanIO stdin                              `thenIO`
       \input -> atm (lines input)                     `thenIO`
       \a -> doneIO

-- The behaviour of an atm specified in monadic operations...

atm :: B ()
atm = insert_card_message          `thenB_`
      card_in                      `thenB_`
      enter_pin_message            `thenB_`
      customer

customer :: B ()
customer =
      pin_no                       `thenB`
      \pin -> validate_pin_no pin  `thenB`
      \valid_pin -> case valid_pin of
       Pin_OK -> service_prompt       `thenB_` services
       Retry  -> retry_message        `thenB_` customer
       Thief  -> keep_card_message    `thenB_` atm
       Wally  -> learn_number_message `thenB_` eject_card `thenB_` atm
       _      -> doneB

services :: B ()
services =
      service                      `thenB`
      \which -> case which of
       Request_Cash       -> cash
       Request_Check_Book -> acknowledge_check_book      `thenB_` more
       Request_Balance    -> show_balance                `thenB_` more
       Request_Statement  -> print_and_proffer_statement `thenB_` more
       _                  -> doneB

cash :: B ()
cash =
      amount_prompt                `thenB_`
      amount                       `thenB`
      \amount_query -> case amount_query of
       Amount_Hopeful -> sorry_but_message `thenB_` cash
       Amount_OK   -> confirm_prompt    `thenB_`
                      confirm           `thenB`
                      \doit -> case doit of
                       Confirm -> proffer_card `thenB_` take_card `thenB `
                                  proffer_cash `thenB_` take_cash `thenB_` atm
                       Cancel  -> more
                       _       -> doneB
       _           -> doneB

more :: B ()
more =
      eject_card                   `thenB_`
```

```
      another_service_message      `thenB_`
      eject                        `thenB`
      \answer -> case answer of
       Continue -> service_prompt  `thenB_` services
       Card_Out -> atm
       _        -> doneB

--

-- messages...

insert_card_message  = message "Please insert your card for service..."

enter_pin_message    = message "Please type your PIN."

keep_card_message    = message "Sorry, too many tries, I'm keeping it!"

learn_number_message = message "Sorry, wrong number!"

retry_message        = message "Incorrect PIN, please try again."

service_prompt       = message "Please select a service 1-4."

acknowledge_check_book = message "A cheque-book will be sent out to you."

show_balance           = message "Your balance is <some-amount>"

print_and_proffer_statement = message "Please take your statement."

amount_prompt     = message "Please type an amount of cash."

sorry_but_message = message "You'll be lucky!"

confirm_prompt    = message "1 to continue, 2 to cancel"

proffer_card      = message "Please take your card."

proffer_cash      = message "Please take your money, have a nice day!"

eject_card        = message "here's yer card."

another_service_message = message "take it with 1, shove it in again with 2."

--

-- hit return...

card_in = hit_return

take_card = hit_return

take_cash = hit_return

-- functions for the "Reply" types...

data Valid_Replys = Pin_OK | Retry | Thief | Wally
data Service_Replys = Request_Cash
                    | Request_Check_Book
                    | Request_Balance
                    | Request_Statement
data Amount_Replys = Amount_Hopeful | Amount_OK
data Confirm_Rsplys = Confirm | Cancel
data Eject_Replys = Continue | Card_Out

int2valid 1 = Pin_OK
```

```
int2valid 2 = Retry
int2valid 3 = Thief
int2valid 4 = Wally

int2service 1 = Request_Cash
int2service 2 = Request_Check_Book
int2service 3 = Request_Balance
int2service 4 = Request_Statement

int2amount n | n < 30     = Amount_OK
             | otherwise = Amount_Hopeful

int2confirm 1 = Confirm
int2confirm 2 = Cancel

int2eject 2 = Continue
int2eject 1 = Card_Out

--

-- functions to take user input into various types...

validate_pin_no :: Int -> B (Valid_Replys)
validate_pin_no x = returnB (int2valid x)

pin_no  = number_input id

service = number_input int2service

amount  = number_input int2amount

confirm = number_input int2confirm

eject   = number_input int2eject

-- some monadic operations...

message :: String -> B ()
message mess =
    appendChanB stdout ("\n"++mess++"\n")

hit_return :: B ()
hit_return =
         getLinesB `thenB`
         \input -> case input of
           Just l -> returnB ()
           Nothing -> doneB

number_input :: (Int -> a) -> B (a)
number_input f =
         getLinesB `thenB`
         \input -> case input of
           Just l -> returnB (f (read l))
           Nothing -> doneB

--

-- Lower-level monad hackery.

type Lines = [String]
data Maybe a = Nothing | Just a

type B a = Lines -> IO (a,Lines)


thenB :: B a -> (a -> B b) -> B b
thenB a k lines =
```

```
        a lines 'thenIO' \(b,lines1) ->
        k b lines1

thenB_ :: B a -> B b -> B b
thenB_ a k lines =
        a lines 'thenIO' \(_,lines1) ->
        k lines1

getLinesB :: B [Maybe String]
getLinesB [] = returnIO (Nothing,[])
getLinesB (l:lines) = returnIO (Just l,lines)

returnB :: a -> B a
returnB a lines = returnIO (a,lines)

promoteB :: IO a -> B a
promoteB io lines
        = io    'thenIO' \a ->
          returnIO (a,lines)

appendChanB x y = promoteB (appendChanIO x y)

doneB = promoteB doneIO

-- "done" in continuations will exit the program.

doneIO = ccall exit 0# 'thenIO_Int#' \ a -> returnIO (error "exit failed?")
```

# Appendix C

# Code and Examples from Chapter 3

## C.1 Swish.c — Extension to Tk

```
/*
 * Copyright 1992-1994 Duncan Sinclair
 *
 * Portions Copyright 1990-1994 Regents of the University of California.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose and without fee is hereby granted, provided
 * that the above copyright notice appear in all copies.  The author make no
 * representation about the suitability of this software for any purpose. It
 * is provided "as is" without express or implied warranty.
 */

/*
 * An extension to allow concurrent communication with an external process.
 * e.g. a Haskell program.
 *
 * Created September 1992
 *
 * Renamed January 1993.
 *
 * Ported to tk 3.1 Feb 1993.
 *
 * Ported to tcl7.3 & tk 3.6 April 1994.
 */

#include "tk.h"

/*
 * Some extra includes...
 */

#include <stdio.h>
#include <ctype.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>

extern int          errno;
```

```
/*
 * Global variables used by swish:
 */

static Tk_Window    w;          /* The main window for the application.  If
                                 * NULL then the application no longer
                                 * exists. */
static Tcl_Interp  *interp;     /* Interpreter for this application. */
static Tcl_DString  combuffer;  /* Used to assemble lines of process input
                                 * into Tcl commands. */


/*
 * Stuff used by my forking process...
 */

/* #define FIFO */

/* their side */
#define C_OUT 13
#define E_IN  14
#define R_IN  15
/* our side */
#define C_IN  13
#define E_OUT 14
#define R_OUT 15

#ifdef FIFO
/* names for named pipes */
char                c[] = "/tmp/.pipec";
char                e[] = "/tmp/.pipee";
char                r[] = "/tmp/.piper";

#endif                          /* FIFO */

static int          cpipe[2];   /* Commands in */
static int          epipe[2];   /* Events out */
static int          rpipe[2];   /* Replies out */
static int          pid;        /* pid of child process (also used as flag) */

/* starting size of length of line */
#define LINE_LEN            200

struct line {
   char              *frep;     /* What to free */
   char              *this;     /* The line */
   int                len;      /* Line length */
   struct line       *next;     /* Next one */
};

struct qhead {
   int                fd;       /* where it's to go */
   int                length;   /* Basically is there anything in queue? */
   struct line       *head;     /* First one - remove from here */
   struct line       *tail;     /* Last one - add after here    */
};

static struct qhead equeue, rqueue;


/*
 * Forward declarations for procedures defined later in this file:
 */

extern int          Swish_Init();
```

```
static int          EventCmd();
static int          ReplyCmd();
static int          ThingCmd();
static int          DummyCmd();
static void         ComProc();

/*
 * Externally visible init routine, called by TkAppInit().
 *
 */
int
Swish_Init(intp)
   Tcl_Interp         *intp;
{
   interp = intp;

   w = Tk_MainWindow(interp);

   Tcl_CreateCommand(interp, "event", EventCmd, (ClientData) NULL,
                     (void (*) ()) NULL);
   Tcl_CreateCommand(interp, "reply", ReplyCmd, (ClientData) NULL,
                     (void (*) ()) NULL);
   Tcl_CreateCommand(interp, "spawnchannels", ThingCmd, (ClientData) NULL,
                     (void (*) ()) NULL);
   Tcl_CreateCommand(interp, "dummy", DummyCmd, (ClientData) NULL,
                     (void (*) ()) NULL);

   Tcl_DStringInit(&combuffer);

   return TCL_OK;
}


/*
 * ComProc takes input coming from "com" channel, and feeds it to the
 * interpreter.
 *
 */

static void
ComProc(clientData, mask)
   ClientData          clientData;      /* Not used. */
   int                 mask;
{
   static int          gotPartial = 0;
   char                line[LINE_LEN];
   char               *cmd;
   int                 ret, result;
   int                 i;

   if (!(mask & TK_READABLE))
      return;

   for (i = 0; i < 10; i++) {
      ret = read(C_IN, line, (LINE_LEN - 1));
      if (ret == -1) {
         if (errno == EINTR) {
            continue;
#if 0
         } else if (errno == EBADF) {
            fputs("Something not right!\n", stderr);
            fflush(stderr);
            Tk_DeleteFileHandler(C_IN);
            break;
#endif
         } else if ((errno == EWOULDBLOCK) || (errno == EAGAIN)) {
            break;
```

```
         } else {                /* something bad's happened... */
            Tcl_Eval(interp, "exit");
            exit(0);
         }
         /* NOTREACHED */
         break;
      }
      if (ret == 0) {            /* is it eof, or is it SysV semantics ?? */
         Tcl_Eval(interp, "exit");
         exit(0);
      }
      line[ret] = '\0';
      cmd = Tcl_DStringAppend(&combuffer, line, ret);
      if (ret != 0) {
         if ((line[ret - 1] != '\n') && (line[ret - 1] != ';')) {
            gotPartial = 1;
            continue;
         }
         if (!Tcl_CommandComplete(cmd)) {
            gotPartial = 1;
            continue;
         }
      }
      Tk_CreateFileHandler(C_IN, 0, ComProc, (ClientData) 0);
      result = Tcl_Eval(interp, cmd);
      Tk_CreateFileHandler(C_IN, TK_READABLE, ComProc, (ClientData) 0);
      Tcl_DStringFree(&combuffer);
#if 1
      /* Not sure what to do with errors yet... */
      if (*interp->result != 0) {
         if ((result != TCL_OK)) {
            printf("%s\n", interp->result);
         }
      }
#endif
   }
}

/*
 * These functions handled queued events and replys to send to the external
 * process.
 *
 */

static void
QueueInit(queue, fd)
   struct qhead       *queue;
   int                 fd;
{
   queue->fd = fd;
   queue->length = 0;
   queue->head = NULL;
   queue->tail = NULL;
}

static int
WriteLine(queue, data)
   struct qhead       *queue;
   struct line        *data;
{
   int                 ret, off;
   char               *p;

   p = data->this;
   off = data->len;
   for (;;) {
      ret = write(queue->fd, p, off);
```

```
        if (ret == -1) {
            if (errno == EINTR) {
                continue;
            } else if ((errno == EWOULDBLOCK) || (errno == EAGAIN)) {
                break;
            } else {                /* must be eof */
                Tcl_Eval(interp, "exit");
                exit(0);
            }
            /* NOTREACHED */
            break;
        } else if (ret == 0) {
            break;
        } else {
            p += ret;
            off -= ret;
            if (off) {
                /* we'll go round again, rather than break this line up */
                continue;
            } else {
                break;
            }
        }
    }
    if (off) {
        /* didn't manage to write it all ! */
        data->this = p;
        data->len = off;
        return 0;
    }
    return 1;
}

static void
FileQueue(clientData, mask)
    ClientData          *clientData;        /* holds the queue */
    int                  mask;
{
    int                  ret;
    struct qhead        *queue = (struct qhead *) clientData;
    struct line         *data = (struct line *) NULL;

    if (!(mask & TK_WRITABLE))
        return;

    for (data = queue->head; data != (struct line *) NULL;) {
        if (!WriteLine(queue, data))
            break;
        /* Great!  Now let's ditch this entry */
        queue->head = data->next;
        (void) free((char *) data->frep);
        (void) free((char *) data);
        queue->length--;
        data = queue->head;
    }
    if ((data == (struct line *) NULL) || (queue->length == 0)) {
        Tk_DeleteFileHandler(queue->fd);
    }
}


static int
WriteQueue(interp, argc, argv, queue)
    Tcl_Interp          *interp;  /* Current interpreter. */
    int                  argc;    /* Number of arguments. */
    char               **argv;    /* Argument strings. */
    struct qhead        *queue;   /* where to queue it. */
```

```
{
    int                 i, off, ret;
    char               *block, *p, *q, *r;
    int                 siz = LINE_LEN;

    block = (char *) malloc(siz * sizeof(char));
    if (block == NULL) {
        Tcl_AppendResult(interp, "out of memory in \"",
                         argv[0], "\".", (char *) NULL);
        return TCL_ERROR;
    }
    p = block;
    r = block + siz;
    /* I love this kind of code! */
    for (i = 1; i < argc; i++) {
        for (q = argv[i]; *p++ = *q++;) {
            if ((r - p) < 4) {
                off = p - block;
                block = realloc(block, siz += 100);
                p = block + off;
                r = block + siz;
                if (block == NULL) {
                    Tcl_AppendResult(interp, "out of memory in \"",
                                     argv[0], "\".", (char *) NULL);
                    return TCL_ERROR;
                }
            }
        }
        *(p - 1) = ' ';
    }
    *(p - 1) = '\n';
    *p = '\0';
    off = p - block;

    /*
     * OK, now that block contains all argv[argc] strung together with spaces
     * between them, what we going to do with it?
     */
    p = block;
    /* if the queue is empty, we'll try writing it straight out. */
    if (queue->length == 0) {
        for (;;) {
            ret = write(queue->fd, p, off);
            if (ret == -1) {
                if (errno == EINTR) {
                    continue;
                } else if ((errno == EWOULDBLOCK) || (errno == EAGAIN)) {
                    break;
                } else {            /* must be eof */
                    Tcl_Eval(interp, "exit");
                    exit(0);
                }
                /* NOTREACHED */
                break;
            } else if (ret == 0) {
                break;
            } else {
                p += ret;
                off -= ret;
                if (off) {
                    /* we'll go round again, rather than break this line up */
                    continue;
                } else {
                    break;
                }
            }
        }
```

```
    }
    if (off) {
        /* something to be queued */
        struct line        *data = (struct line *) malloc(sizeof(struct line));

        if (data == NULL) {
            Tcl_AppendResult(interp, "out of memory in \"",
                             argv[0], "\".", (char *) NULL);
            return TCL_ERROR;
        }
        if (!queue->length) {
            /* first entry, register file handler */
            Tk_CreateFileHandler(queue->fd, TK_WRITABLE,
                                 FileQueue, (ClientData) queue);
        }
        data->rep = block;
        data->this = p;
        data->len = off;
        data->next = (struct line *) NULL;

        /* OK, here's the fun bit - add the sucker to the end of the queue! */
        queue->length++;
        if (queue->head == (struct line *) NULL)
            queue->head = data;
        else
            queue->tail->next = data;
        queue->tail = data;
    } else {
        (void) free((char *) block);
    }

    return TCL_OK;
}

/*
 * EventCmd & ReplyCmd grab their args, stick 'em together, and try to send
 * them off to the other side.  If this blocks, then we stick 'em on the end
 * of a queue, and worry about them another time.
 */

static int
EventCmd(dummy, interp, argc, argv)
    ClientData          dummy;    /* Not used. */
    Tcl_Interp         *interp;   /* Current interpreter. */
    int                 argc;     /* Number of arguments. */
    char               **argv;    /* Argument strings. */
{
    if (!pid) {
        Tcl_AppendResult(interp, "No co-process currently running, in \"",
                         argv[0], "\".", (char *) NULL);
        return TCL_ERROR;
    }
    return WriteQueue(interp, argc, argv, &equeue);
}

static int
ReplyCmd(dummy, interp, argc, argv)
    ClientData          dummy;    /* Not used. */
    Tcl_Interp         *interp;   /* Current interpreter. */
    int                 argc;     /* Number of arguments. */
    char               **argv;    /* Argument strings. */
{
    if (!pid) {
        Tcl_AppendResult(interp, "No co process currently running, in \"",
                         argv[0], "\".", (char *) NULL);
        return TCL_ERROR;
    }
```

```
    return WriteQueue(interp, argc, argv, &rqueue);
}

static int
makepipes()
{
#ifdef FIFO
    unlink(c);
    unlink(e);
    unlink(r);
    if ((mkfifo(c, 0600) == -1) || (mkfifo(e, 0600) == -1) ||
        (mkfifo(r, 0600) == -1))
        goto error;
    if ((cpipe[0] = open(c, O_RDONLY | O_NDELAY)) == -1)
        goto error;
    if ((epipe[0] = open(e, O_RDONLY | O_NDELAY)) == -1)
        goto error;
    if ((rpipe[0] = open(r, O_RDONLY | O_NDELAY)) == -1)
        goto error;
    if ((cpipe[1] = open(c, O_WRONLY)) == -1)
        goto error;
    if ((epipe[1] = open(e, O_WRONLY)) == -1)
        goto error;
    if ((rpipe[1] = open(r, O_WRONLY)) == -1)
        goto error;

    return 0;
error:
    unlink(c);
    unlink(e);
    unlink(r);
    return -1;

#else                           /* FIFO */
    if ((pipe(cpipe) == -1) || (pipe(epipe) == -1) || (pipe(rpipe) == -1)) {
        return -1;
    }
#endif                          /* FIFO */
}

static void
CloseEm()
{
    if (cpipe[0] > 0)
        close(cpipe[0]);
    if (cpipe[0] > 0)
        close(epipe[0]);
    if (rpipe[0] > 0)
        close(rpipe[0]);
    if (cpipe[1] > 0)
        close(cpipe[1]);
    if (epipe[1] > 0)
        close(epipe[1]);
    if (rpipe[1] > 0)
        close(rpipe[1]);
}

/*
 * Split, fork, etc...
 *
 * The 'spawnchannels' command. (a.k.a 'thing')
 */

static int
ThingCmd(dummy, interp, argc, argv)
    ClientData          dummy;  /* Not used. */
    Tcl_Interp         *interp; /* Current interpreter. */
```

```
    int                 argc;     /* Number of arguments. */
    char               **argv;     /* Argument strings. */
{
    int                 tempfd;

    if (argc <= 1) {
        Tcl_AppendResult(interp, "wrong # args: should be \"", argv[0],
                " program <args>\"", (char *) NULL);
        return TCL_ERROR;
    }
    if (pid) {
        Tcl_AppendResult(interp, "co-process currently running, in \"",
                argv[0], "\".", (char *) NULL);
        return TCL_ERROR;
    }
    if (makepipes() == -1) {
        CloseEm();
        Tcl_AppendResult(interp, "couldn't create pipes in \"",
                argv[0], "\".", (char *) NULL);
        return TCL_ERROR;
    }
    if ((pid = fork()) == -1) {
        CloseEm();
        Tcl_AppendResult(interp, "couldn't fork in \"",
                argv[0], "\".", (char *) NULL);
        return TCL_ERROR;
    } else if (pid == 0) {        /* child */
        if ((dup2(cpipe[1], C_OUT) == -1) || (dup2(epipe[0], E_IN) == -1) ||
            (dup2(rpipe[0], R_IN) == -1)) {
            perror("dups");
            _exit(2);
        }
        CloseEm();
        /* if (fileName != NULL) { */
        close(0);
        if ((tempfd = open("/dev/null", O_RDONLY)) != 0) {        /* Arg: */
            close(tempfd);          /* give up */
        }
        /* } */
        if (w != NULL)
            close(XConnectionNumber(Tk_Display(w)));
        execvp(argv[1], &argv[1]);
        perror("execvp");
        _exit(3);
    } else {                        /* parent */
        if ((dup2(cpipe[0], C_IN) == -1) || (dup2(epipe[1], E_OUT) == -1) ||
            (dup2(rpipe[1], R_OUT) == -1)) {
            pid = 0;
            CloseEm();
            close(C_IN);
            close(E_OUT);
            close(R_OUT);
            Tcl_AppendResult(interp, "couldn't dup in \'",
                             argv[0], "\'.", (char *) NULL);
            return TCL_ERROR;
        }
        CloseEm();
        fcntl(C_IN, F_SETFL, O_NDELAY);
        fcntl(E_OUT, F_SETFL, O_NDELAY);
        fcntl(R_OUT, F_SETFL, O_NDELAY);
    }
    Tk_CreateFileHandler(C_IN, TK_READABLE, ComProc, (ClientData) 0);
    QueueInit(&equeue, E_OUT);
    QueueInit(&rqueue, R_OUT);
    /* Tcl_DetachPids(1, &pid); */

    /*
```

```
    * Any other initialisations here...
    */
   return TCL_OK;
}

static int
DummyCmd(dummy, interp, argc, argv)
   ClientData          dummy;    /* Not used. */
   Tcl_Interp         *interp;   /* Current interpreter. */
   int                 argc;     /* Number of arguments. */
   char               **argv;    /* Argument strings. */
{
   return TCL_OK;
}
```

## C.2 Alarm Clock — Tcl Code

```
#!/usr/X11/local/bin/swish -f
#
# Front-end to a Haskell Clock
#

# Some variables...

set alarmed 0
set setting 0

proc mainwindow {} {
    frame .top -relief raised -border 1
    frame .bot -relief raised -border 1
    pack append . .top {top fill expand} .bot {top fill expand}

    message .top.info -text "The time sponsored by Haskell is" \
    -justify center -aspect 1200  -font -*-times-medium-i-*--*-240-*
    message .top.time -justify center -aspect 1200  \
            -font -*-times-medium-i-*--*-240-*
    pack append .top .top.info {top padx 10 pady 10 expand}
    pack append .top .top.time {top padx 10 pady 10 expand}

    button .bot.alarm -text "Set Alarm" -command {setalarm}
    button .bot.death -text "Out Of Time" -command {destroy .}
    pack append .bot .bot.alarm {left expand padx 20 pady 20}
    pack append .bot .bot.death {left expand padx 20 pady 20}
}

proc alarm {} {
    global alarmed

    if {$alarmed} {return}
    set alarmed 1

    toplevel .alarm

    button .alarm.button -text "Alarm!!!" -command "desalarm" \
                         -font -*-charter-bold-r-*--*-240-*
    pack append .alarm .alarm.button {expand padx 30 pady 30}
    after 200 flasher
}

proc flasher {} {
    global alarmed

    if {[expr !$alarmed]} {return}

    .alarm.button flash
    after 500 flasher
}

proc setalarm {} {
    global setting

    if {$setting} {return}
    set setting 1

    desalarm

    toplevel .setter
    frame .setter.top -relief raised -border 1
    frame .setter.bot -relief raised -border 1
    pack append .setter .setter.top {top fill expand} \
```

```
                        .setter.bot {top fill expand}

    message .setter.top.text -text "Set alarm for HH:MM:SS" -aspect 1200
    entry .setter.top.time -relief sunken
    bind .setter <Return> "getalarm"
    bind .setter.top.time <Return> "getalarm"

    pack append .setter.top \
              .setter.top.text {top pady 10 fillx} \
              .setter.top.time {top padx 10 pady 10 expand}

    button .setter.bot.ok -text OK -command 'getalarm"
    pack append .setter.bot  .setter.bot.ok {expand padx 20 pady 20}
}

proc getalarm {} {
    global setting

    if {[expr !$setting]} {return}
    set tim [.setter.top.time get]
    if {[regexp {[0-9][0-9]:[0-9][0-9]:[0-9][0-9]} $tim]} {
      event $tim
      destroy .setter
      set setting 0
     }
}

proc desalarm {} {
    global alarmed

    if {$alarmed} {
        destroy .alarm
        set alarmed 0
    }
}

proc disp {args} {
    .top.time configure -text $args
}

# create main window...
mainwindow

# run the program...
spawnchannels bcc
```

## C.3 Alarm Clock — Haskell Code

```
module Main(main) where

import Time

type Chans = [(String,Char)]

main = openevents $
       \events -> openreply $
       \reply  -> process (cpipe,events,reply,"","")


process :: Result
process = getevent    $
        \event -> case event of
          (c,x) | c == cpipe -> dochar x $ process
          (c,x) | c == ticker -> dotime $ process
          _ -> mydone


type Result = (String,Chans,Chans,String,String) -> [Response] -> [Request]

getevent :: ((String,Char) -> Result) -> Result
getevent cont =
  \(chan,(a:events),replys,alrm,new) ->
    cont a (chan,events,replys,alrm,new)

dochar :: Char -> Result -> Result
dochar x cont =
  \(chan,events,replys,alrm,new) ->
    if (x == chr (-1)) then -- eof
      mydone (chan,events,replys,alrm,new)
    else
      case x of
        '\n' -> (cont (chan,events,replys,new,""))
        _    -> (cont (chan,events,replys,alrm,(new ++ [x])))

dotime :: Result -> Result
dotime cont =
  \(chan,events,replys,alrm,new) ->
    getLocalTime (die "dotime") $
    \time -> if ((tss time) == alrm) then
        appendChan chan "alarm\n" (die "alarm") $
        appendChan chan (ts time) (die "dotimea") $
        cont (chan,events,replys,alrm,new)
      else
        appendChan chan (ts time) (die "dotimea") $
        cont (chan,events,replys,alrm,new)


mydone :: Result
mydone _ = done

continuation :: (b -> a) -> b -> (a -> Result) -> Result
continuation x y z = z (x y)


openevents = readChannels [cpipe,ticker] (die "opene")

openreply  = readChannels [rpipe,tmout] (die "openr")


-- bits
```

```
cpipe = "/dev/fd13"
epipe = "/dev/fd14"
rpipe = "/dev/fd15"

ticker = "TICK:1"
tmout = "TIMEOUT:5"

ts num = "disp " ++ ((timeToStr . dblToTime) num) ++ "\n"

tss = timeTotStr . dblToTime

show2 :: Int -> String
show2 x = [chr (x `quot` 10 + ord '0'), chr (x `rem` 10 + ord '0')]

weekdays = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"]

months = ["Jan","Feb","Mar","Apr","May","Jun",
          "Jul","Aug","Sep","Oct","Nov","Dec"]

th  1 = "st"
th  2 = "nd"
th  3 = "rd"
th 21 = "st"
th 22 = "nd"
th 23 = "rd"
th 31 = "st"
th  x = "th"

timeTotStr :: Time -> String
timeTotStr (Time year mon day hour min sec sdec wday) =
    show2 hour ++ ":" ++ show2 min ++ ":" ++ show2 sec

timeToStr :: Time -> String
timeToStr (Time year mon day hour min sec sdec wday) =
    show2 hour ++ ":" ++ show2 min ++ ":" ++ show2 sec ++ " " ++
    weekdays!!wday ++ " " ++ months!!mon ++ " " ++ show day ++
    th day  ++ " " ++ show year

type TagCont = Chars -> Dialogue

readChannels :: [String] -> FailCont -> TagCont -> Dialogue
readChannels list fail succ resps =
    (ReadChannels list) : tagDispatch fail succ resps

tagDispatch fail succ (resp:resps) =
            case resp of
             Tag val     -> succ val resps
             Failure msg -> fail msg resps


die what ( WriteError foo) =
         appendChan stderr (what ++ ": write: " ++ foo ++ "\n") abort done
die what (  ReadError foo) =
         appendChan stderr (what ++ ": read: " ++ foo ++ "\n") abort done
die what (SearchError foo) =
         appendChan stderr (what ++ ": search: " ++ foo ++ "\n") abort done
die what (FormatError foo) =
         appendChan stderr (what ++ ": format: " ++ foo ++ "\n") abort done
die what ( OtherError foo) =
         appendChan stderr (what ++ ": error: " ++ foo ++ "\n") abort done
```

## C.4 Functional Maze in X — Tcl Code

```
#!swish -f
#
# Copyright 1992,1993 Duncan Sinclair
#

set knotlib "swish"

set amaze "amaze"
set icon  "@maze.icon"

#  large window...
set area 600
set block 550
set eye 1000
set gap 150

set depth 5

#light
set c1 #ffa500
#lighter
set c2 #faa200
#darker
set c3 #f59f00
#dark
set c4 #f09c00

set col0 $c1
set col0a ""

set col1 $c3
set col2 $c3
set col3 $c3
set col4 $c3

set col5 $c2
set col6 $c2
set col7 $c2
set col8 $c2

set col9 $c4

proc mainwindow {} {
        global area depth block

        frame .buttons -relief raised -border 2 -width 30
        frame .view -relief raised -border 2
        pack append . .view {left fill expand} .buttons {left fill expand}

        frame .buttons.grid1
        frame .buttons.grid2
        frame .buttons.grid3
        frame .buttons.grid4
        pack append .buttons \
                .buttons.grid1 {top fillx} \
                .buttons.grid2 {top fillx} \
                .buttons.grid3 {top fillx} \
                .buttons.grid4 {top fill expand}

        button .buttons.grid1.forw -text "Forward" \
                -command "event in" -height 3
        button .buttons.grid2.left -text "Left" \
                -command "event left" -height 3
```

```
        button .buttons.grid2.righ -text "Right" \
                -command "event right" -height 3
        button .buttons.grid3.back -text "Back Up" \
                -command "event out" -height 3

        pack append .buttons.grid1 \
                .buttons.grid1.forw {top padx 20 pady 20 fill expand}

        pack append .buttons.grid2 \
                .buttons.grid2.left {left padx 20 pady 20 fill expand} \
                .buttons.grid2.righ {left padx 20 pady 20 fill expand}

        pack append .buttons.grid3 \
                .buttons.grid3.back {bottom padx 20 pady 20 fill expand}

        button .buttons.grid4.start -text "Restart" \
                -width 20 -command "event init"
        button .buttons.grid4.new -text "New Game" \
                -width 20 -command "event new"
        button .buttons.grid4.quit -text "Quit" \
                -width 20 -command "destroy ."

        pack append .buttons.grid4 \
                .buttons.grid4.quit  {bottom padx 20 pady 20} \
                .buttons.grid4.new   {bottom padx 20 pady 20} \
                .buttons.grid4.start {bottom padx 20 pady 20}

        canvas .view.piccy -relief raised -border 1 -height $area -width $area
        message .view.status -text "" -aspect 2000 -relief sunken -border 1

        pack append .view .view.piccy {padx 20 pady 20}
        pack append .view .view.status {bottom fill}

        bind . h "event left"
        bind . j "event out"
        bind . k "event in"
        bind . l "event right"
}

proc drawrects {} {
        global area depth block eye gap
        global col0 col0a col1 col2 col3 col4
        global col5 col6 col7 col8 col9

        set can .view.piccy

        $can create rectangle 0 0 $area $area -width 0 -fill $col9
        set centre [expr "$area / 2"]
        set opersp $centre
        set opersp [expr "( $centre * $eye ) /  \
                ( $eye + $gap - ( $depth * $block ) )"]
        set c [expr "$centre - $opersp"]
        set d [expr "$centre + $opersp"]
        $can create rectangle $c $c $d $d -width 0 -fill white
        $can create rectangle $c $c $d $d -width 0 -fill black -stipple gray50
        set s 0
        set t $area
        for {set i $depth} {$i>=0} {set i [expr "$i - 1"]} {
                set persp [expr "($centre * $eye)/ \
                        ($eye + $gap + (($i - 1) * $block))"]
                if {[expr "$i == 0"]} {
                set persp $centre
                }

                set a [expr "$centre - $persp"]
                set b [expr "$centre + $persp"]
                set c [expr "$centre - $opersp"]
```

```
                set d [expr "$centre + $opersp"]

                set a1 [expr "$a + 1"]
                set a2 [expr "$a + 2"]
                set b1 [expr "$b - 1"]
                set b2 [expr "$b - 2"]
                set c1 [expr "$c - 1"]
                set d1 [expr "$d + 1"]

                # back walls
                $can create polygon $c $c $c $d $d $d $d $c -fill $col0 -tags a0$i

                # 4 walls ajoining back wall.
                $can create polygon $c $s $c $c $d $c $d $s  -fill $col1   -tags a1$i
                $can create polygon $t $c $d $c $d $d $t $d  -fill $col2   -tags a2$i
                $can create polygon $d $t $d $d $c $d $c $t  -fill $col3   -tags a3$i
                $can create polygon $s $d $c $d $c $c $s $c  -fill $col4   -tags a4$i

                # border lines of this layer's walls.
                $can create line $a $a $c $c $c $s $c $c $d $c $d $s $d $c $b $a
                $can create line $b $a $d $c $t $c $d $c $d $d $t $d $d $d $b $b
                $can create line $b $b $d $d $d $t $d $d $c $d $c $t $c $d $a $b
                $can create line $a $b $c $d $s $d $c $d $c $c $s $c $c $c $a $a

                # 4 side walls.
                $can create polygon $a2 $a1  $c $c  $d $c  $b1 $a1 \
                        -fill $col5 -tags a5$i
                $can create polygon $b $a1  $d1 $c  $d1 $d  $b $b1 \
                        -fill $col6 -tags a6$i
                $can create polygon $b $b  $d $d  $c $d1  $a1 $b \
                        -fill $col7 -tags a7$i
                $can create polygon $a $b1  $c $d  $c $c  $a $a \
                        -fill $col8 -tags a8$i

                set cpersp $persp
        }
}

#  "public" proceedures...

# set a wall on...
proc on {depth wall} {
        global col0 col1 col2 col3 col4 col5 col6 col7 col8 col9

        append aa a $wall $depth
        append bb col $wall
        .view.piccy itemconfigure $aa -fill [set $bb]
}

#set a wall off...
proc off {depth wall} {
        append aa a $wall $depth
        .view.piccy itemconfigure $aa -fill ""
}

# set the walls for a particular depth...
proc walls {depth args} {
        global col0 col1 col2 col3 col4 col5 col6 col7 col8 col9

        set wall 0
        foreach foo $args {
                set aa ""
                set bb ""
                append aa a $wall $depth
                case $foo in {
                  {0}
                        {.view.piccy itemconfigure $aa -fill ""}
```

```
                    {1}
                        {
                          append bb col $wall
                          .view.piccy itemconfigure $aa -fill [set $bb]
                        }
                    }
                    set wall [expr [$wall + 1]]
            }
}

#set info message...
proc status {a args} {
    set mess ""
    case $a in {
     {b}
      {append mess "Hey you've been here before.  You're still missing " \
   $args \
   " rooms."}
     {w}
      {append mess "Welcome to the Maze!"}
     {f}
      {append mess "You've now seen all the rooms.  It took you " \
   $args \
   " moves."}
     {m}
      {append mess "You're still missing " $args " rooms."}
    }
    .view.status configure -text $mess
}


wm iconbitmap . $icon
mainwindow
status Welcome to the Maze!!
drawrects

spawnchannels $amaze $argv
```

## C.5 Functional Maze in X — Haskell Code

### Amaze.hs

```
module Main(main) where

import Maze

{--
---     Maze constants
--}


mazeDimenX = 4
mazeDimenY = 4
mazeDimenZ = 4
seed = 987

{--
---     Main
--}

main = getArgs exit (\argv ->
       let arg = case argv of [w] -> words w; _ -> []
           [s,x,y,z] = if length arg == 4
                       then (map read arg :: [Int])
                       else [seed,mazeDimenX,mazeDimenY,mazeDimenZ]
       in
           readChan epipe exit
              (\inC -> newMaze (davidsRandoms s) x y z ("init":lines inC)))

display = displayMaze

newMaze (s:ss) x y z events =
   let maze = makeMaze s x y z
       mazeDepth = min 8 (maximum [x,y,z])

       mainLoop h m []     = done
       mainLoop h m (i:r) =
         case i of
           "new"   -> newMaze ss x y z r
           "init"  ->
              toCpipe (display maze mazeDepth)
              (toCpipe "status w\n" -- "w"elcome
              (mainLoop (0,[position maze]) maze r))
           "in"    -> mainLoop' (hsucc h) (m,(walkForward m)) r
           "left"  -> mainLoop' (hsucc h) (m,(turnLeft m)) r
           "right" -> mainLoop' (hsucc h) (m,(turnRight m)) r
           "out"   -> mainLoop' (hsucc h) (m,(walkBack m)) r
           other   ->
              toErr ("Unknown instruction: "++other++"\n")
              (mainLoop h m r)

       mainLoop' h (m0,m1) r
         =  toCpipe (display m1 mazeDepth)
            (areWeThere h (m0,m1) r)

       areWeThere (moves,seen) (m0,m1) r
         = if position m0 == position m1
           then {- we are just looking around -}
              mainLoop (moves,seen) m1 r
           else {- we are in a new room, Maybe! -}
           if position m1 `elem` seen
           then {- we've been here before -}
```

```
            haunt (moves,seen) m1 r
          else {- this is new - maybe we are lost :-) -}
            congratulate (moves,seen) m1 r

      haunt (moves,seen) m r
        = toCpipe (hauntString (x*y*z-(length seen)))
          (mainLoop (moves,seen) m r)

      congratulate (moves,seen) m r
        = let nseen = position m : seen
              next = mainLoop (moves,nseen) m r
          in if (length nseen) == (x*y*z)
             then {- we've seen it all -}
               toCpipe (seenAllString moves) next
             else {- we still missing some rooms -}
               toCpipe (gettingThereString (x*y*z-(length nseen))) next

      hauntString n -- "b"een here before.
        = "status b " ++ show n ++ "\n"

      seenAllString n  -- "f"inished the maze
        = "status f " ++ show n ++ "\n"
          ++ "It took you " ++ show n ++ " moves.\n"

      gettingThereString n -- "m"issing rooms
        = "status m " ++ show n ++ "\n"

  in
      mainLoop emptyHistory maze ("init":events)

toCpipe s = appendChan cpipe s exit
toErr   s = appendChan stderr s exit
toOut   s = appendChan stdout s exit


emptyHistory = (0,[])
hsucc (n,s) = (n+1,s)
position (walls,orienv,oriens,pos,size) = pos

{--
---  Implementation constants
--}

-- these are the correct definitions for hbc v. 0.999.{1,2}
cpipe = "/dev/fd13"
epipe = "/dev/fd14"
rpipe = "/dev/fd15"
```

## Maze.hs

```
module Maze where

{--
---     A maze i represented by three 'arrays' of walls
---     (leftWalls,downWalls,backWalls)
---
---     the orientation of the beetle is represented by three
---     vector selector functions
---     (right,up,front)
---
---     Lastly there is the position of the beetle in the array
---     and the dimensions of the array
---
```

```
--}


{--
---     Turning in a Maze (haze?)
---     We are missing the rollLeft, and rollRight
--}

turnRight (walls,(rv,uv,fv),(rs,us,fs), pos, size)
  = (walls, (vneg fv, uv, rv), (fs, us, rs), pos, size)
turnLeft (walls,(rv,uv,fv),(rs,us,fs), pos, size)
  = (walls, (fv, uv, vneg rv), (fs, us, rs), pos, size)
turnUp (walls,(rv,uv,fv),(rs,us,fs), pos, size)
  = (walls, (rv, vneg fv, uv), (rs, fs, us), pos, size)
turnDown (walls,(rv,uv,fv),(rs,us,fs), pos, size)
  = (walls, (rv, fv, vneg uv), (rs, fs, us), pos, size)

{--
---     Let's Move
--}

moveRel (x,y,z) (walls, (rv,uv,fv), (rs,us,fs), pos, size)
  = let rpos = (vadd (vsca x rv)
                (vadd (vsca y uv)
                (vadd (vsca z fv) pos)))
    in (walls, (rv,uv,fv), (rs,us,fs), rpos, size)

moveLeft = moveRel (-1,0,0)
moveRight = moveRel (1,0,0)
moveUp = moveRel (0,1,0)
moveDown = moveRel (0,-1,0)
moveForward = moveRel (0,0,1)
moveBack = moveRel (0,0,-1)

{--
---     Let's do them Beetle moves.
---     Shake 'em, shake 'em.
--}

walkForward maze =
   if frontWall (0,0,0) maze then turnUp maze      else
   if downWall (0,0,1) maze  then moveForward maze else
   if backWall (0,-1,1) maze then turnDown (moveDown (moveForward maze))
   else {- that wall has to be there we are standing on it -}
        turnDown (turnDown (moveDown maze))

walkBack = turnLeft . turnLeft . walkForward . turnLeft . turnLeft

{--
---     Wall peeping
--}

lookAtFrontWall (walls, (rv, uv, fv), (rs,us,fs), pos, size)
  = let cpos = vadd pos (vhalf (vadd (vabs fv) fv))
    in (fs walls) cpos

leftWall r = lookAtFrontWall . turnLeft . moveRel r
rightWall r = lookAtFrontWall . turnRight . moveRel r
downWall r =  lookAtFrontWall . turnDown . moveRel r
upWall r = lookAtFrontWall . turnUp . moveRel r
backWall r = lookAtFrontWall . turnLeft . turnLeft . moveRel r
frontWall r = lookAtFrontWall . moveRel r


{--
---     Maze Creation.
--}
```

```
makeMaze seed a b c =
   let allR = fullMazeRooms a b c
       allW = shuffle seed (fullMazeWalls a b c)
       demolishedWalls = fst3 (iterate connect ([],allR,allW)!!(a*b*c-1))

       lWA = array ((2,1,1),(a,b,c))
             [(x,y,z) := ((x-1,y,z),(x,y,z)) `notElem` demolishedWalls |
              x <- [2..a], y <- [1..b], z <- [1..c]]
       dWA = array ((1,2,1),(a,b,c))
             [(x,y,z) := ((x,y-1,z),(x,y,z)) `notElem` demolishedWalls |
              x <- [1..a], y <- [2..b], z <- [1..c]]
       bWA = array ((1,1,2),(a,b,c))
             [(x,y,z) := ((x,y,z-1),(x,y,z)) `notElem` demolishedWalls |
              x <- [1..a], y <- [1..b], z <- [2..c]]

       lW (x,y,z) | inRange ((2,1,1),(a,b,c)) (x,y,z) = lWA!(x,y,z)
                  | True                              = True
       dW (x,y,z) | inRange ((1,2,1),(a,b,c)) (x,y,z) = dWA!(x,y,z)
                  | True                              = True
       bW (x,y,z) | inRange ((1,1,2),(a,b,c)) (x,y,z) = bWA!(x,y,z)
                  | True                              = True

       pos0 = (1,1,1)
       orienv = ((1,0,0),(0,1,0),(0,0,1))
       oriens = (fst3,snd3,thd3)
    in ((lW,dW,bW), orienv, oriens, pos0, (a,b,c))

connect (ws,rooms,(w:rw)) =
  let (r1,r2) = w
      (connected,nrooms) = isConnectedRooms rooms r1 r2
  in if not connected
     then {  Good Wall -}
          (w:ws,nrooms,rw)
     else {- Bad Wall, try again -}
          connect (ws,nrooms,rw)

{--
---     Connecting equivalence classes
--}

isConnectedRooms rooms r1 r2 =
   let connected = rooms r1 == rooms r2
       nrooms = connectRooms rooms r1 r2
   in if connected then (connected,rooms)
                   else (connected,nrooms)

{- We start out with all rooms unconnected -}
fullMazeRooms a b c = \(x,y,z) -> ((x-1)*b + (y-1))*c + (z-1)

{- later we connect the rooms one by one -}
connectRooms rooms a b =
   let ra = rooms a
       rb = rooms b
   in (\r -> let rr = rooms r in if rr == ra then rb else rr)

{- lets build some walls -}

fullMazeWalls a b c =
  [((x,y,z),(x+d1,y+d2,z+d3))
  | (d1,d2,d3) <- [(1,0,0),(0,1,0),(0,0,1)],
    x <- [1..(a-d1)],
    y <- [1..(b-d2)],
    z <- [1..(c-d3)]]

{--
---     Shuffle a list so the elements come in random order
```

```
---     Make sure that we don't have to be in the same Maze all the time
--}

shuffle :: Int -> [a] -> [a]
shuffle seed =
   let leq :: (Int,a) -> (Int,a) -> Bool
       leq (x,_) (y,_) = x <= y
   in  map snd .
       quicksort leq .
       zip (davidsRandoms seed)

{- David Lesters random numbers -}

davidsRandoms :: Int -> [Int]
davidsRandoms = filter (/= (m-1)) .
                iterate (\seed-> (a * seed + c) `mod` m)
                where m = 65537
                      a = 272
                      c = 2113

{- The standard Quicksort - or is it more like shell short -}

quicksort leq [] = []
quicksort leq (m:xs) =
   quicksort leq [x|x<-xs, x `leq` m]
   ++ [m] ++
   quicksort leq [x|x<-xs, not(x `leq` m)]


{--
---     Make a nice picture of the maze
--}

printMaze m =
   let (w,ov,os,p,(a,b,c)) = m
       maze = (w,ov,os,(0,0,0),(a,b,c))
       frontW y z =
         "+"++
         concat[if frontWall (x,y,z) maze then "-+" else " +"
                | x <- [1..a]]++"\n"
       leftWallAndFloor y z =
          concat[ (if leftWall (x,y,z) maze then "|" else " ") ++
                  (if downWall (x,y,z) maze then " " else "*")
                | x <- [1..a]]++(if rightWall (a,y,z) maze
                                 then "|\n" else "\n")
       layer y =
         concat [ frontW y z ++ leftWallAndFloor y z | z <- [c,(c-1)..1]] ++
         "+"++concat[if backWall (x,y,1) maze then "-+" else " +"
                     | x <- [1..a]]++"\n\n"
   in concat [ layer y | y <- [b,(b-1)..1]]



{--
---     Set the walls out into the distance.
--}

displayMaze m displayDepth =
   let setOnOff b i j =
         (if b then "on " else "off ")
         ++ show i ++ " " ++ show j ++ "\n"
       setLevel i =
         setOnOff (frontWall (0,0,i) m)  i 0 ++
         setOnOff (frontWall (0,1,i) m)  i 1 ++
         setOnOff (frontWall (1,0,i) m)  i 2 ++
         setOnOff (frontWall (0,-1,i) m) i 3 ++
         setOnOff (frontWall (-1,0,i) m) i 4 ++
```

```
        setOnOff (upWall    (0,0,i) m)  i 5 ++
        setOnOff (rightWall (0,0,i) m)  i 6 ++
        setOnOff (downWall  (0,0,i) m)  i 7 ++
        setOnOff (leftWall  (0,0,i) m)  i 8
   in concat [ setLevel i | i <- [0..displayDepth]]



{--
---     Change the Walls - save time -
--}

displayDiffMaze (m0,m1) displayDepth =
   let setDiffOnOff w r i j =
         if w r m0 /= w r m1 then setOnOff (w r m1) i j else ""
       setOnOff b i j =
          (if b then "on " else "off ")
          ++ show i ++ " " ++ show j ++ "\n"
       setLevel i =
         setDiffOnOff frontWall (0,0,i)  i 0 ++
         setDiffOnOff frontWall (0,1,i)  i 1 ++
         setDiffOnOff frontWall (1,0,i)  i 2 ++
         setDiffOnOff frontWall (0,-1,i) i 3 ++
         setDiffOnOff frontWall (-1,0,i) i 4 ++
         setDiffOnOff upWall    (0,0,i)  i 5 ++
         setDiffOnOff rightWall (0,0,i)  i 6 ++
         setDiffOnOff downWall  (0,0,i)  i 7 ++
         setDiffOnOff leftWall  (0,0,i)  i 8
   in concat [ setLevel i | i <- [0..displayDepth]]

displayLevels m displayDepth =
  let setOnOff b = if b then " 1" else " 0"
      setLevel i =
         "walls " ++ show i ++
         setOnOff (frontWall (0,0,i) m)   ++
         setOnOff (frontWall (0,1,i) m)   ++
         setOnOff (frontWall (1,0,i) m)   ++
         setOnOff (frontWall (0,-1,i) m)  ++
         setOnOff (frontWall (-1,0,i) m)  ++
         setOnOff (upWall    (0,0,i) m)   ++
         setOnOff (rightWall (0,0,i) m)   ++
         setOnOff (downWall  (0,0,i) m)   ++
         setOnOff (leftWall  (0,0,i) m)   ++ "\n"
   in concat [ setLevel i | i <- [0..displayDepth]]



amaze seed a b c = printMaze (makeMaze seed a b c)


{--
---     Primitive operations used.
--}


{- zipWith -}
v3z p (x,y,z) (a,b,c) = (p x a, p y b, p z c)
{-map-}
v3m p (x,y,z) = (p x, p y, p z)
{-foldr-}
v3f p u (x,y,z) = p x (p y (p z u))

type Int3 = (Int,Int,Int)

vadd :: Int3 -> Int3 -> Int3
vadd = v3z (+)
```

```
vsca :: Int -> Int3 -> Int3
vsca = \s -> v3m (s*)

vabs :: Int3 -> Int3
vabs = v3m abs

vneg :: Int3 -> Int3
vneg = v3m (negate)

vhalf :: Int3 -> Int3
vhalf = v3m (`div` 2)

fst3 (x,_,_) = x
snd3 (_,x,_) = x
thd3 (_,_,x) = x
```

## C.6 Minimal Interactive Program — Tcl Code

```
# Interface for Minimally Interactive Program.

# A single global variable.

set asked 0

proc mainwindow {} {
  frame .top -relief raised -border 1
  frame .bot -relief raised -border 1
  pack append . .top {top fill expand} .bot {top fill expand}

  message .top.info -text "This is a minimal interactive program." \
    -justify center -aspect 1200  -font -*-times-medium-i-*--*-240-*
  pack append .top .top.info {top padx 10 pady 10 expand}

  button .bot.death -text "Quit" -command {destroy .}
  pack append .bot .bot.death {left expand padx 20 pady 20}
}

proc ask {text} {
  global asked

  if {$asked} {return}
  set asked 1

  toplevel .ask

  frame .ask.top -relief raised -border 1
  frame .ask.mid -relief raised -border 1
  frame .ask.bot -relief raised -border 1

  pack append .ask .ask.top {top fill expand} \
   .ask.mid {top fill expand} \
   .ask.bot {top fill expand}

  message .ask.top.question -text $text \
    -justify center -aspect 1200  -font -*-times-medium-i-*--*-240-*

  pack append .ask.top .ask.top.question {top padx 10 pady 10 expand}


  button .ask.mid.yes -text "Yes :-)" -command "yes" \
-font -*-charter-bold-r-*--*-240-*
  button .ask.mid.no -text "No :-(" -command "no" \
-font  *-charter-bold-r-*--*-240-*

  pack append .ask.mid .ask.mid.yes {expand padx 30 pady 30}
  pack append .ask.mid .ask.mid.no {expand padx 30 pady 30}


  message .ask.bot.answer -text "" \
    -justify center -aspect 1200  -font -*-times-medium-i-*--*-240-*

  pack append .ask.bot .ask.bot.answer {top padx 10 pady 10 expand}
}

proc yes {} {
  event yes
}

proc no {} {
  event no
}
```

```
proc answer {text} {
   .ask.bot.answer configure -text $text
}

# create main window...
mainwindow

# run main program...
spawnchannels how
```

## C.7 ATM — Tcl Code

```
#!/usr/X11/local/bin/swish -f
#
# Copyright 1995 Duncan Sinclair
#


#  Small window...
#set area 400
#set block 350
#set eye 800
#set gap 150
#

set buffer ""

proc mainwindow {} {

    message .title -aspect 2000 \
   -text "Sinclair Bank ATM"
    pack append . .title {top fill}

    frame .view    -relief raised -border 2 -width 200
    frame .buttons -relief raised -border 2 -width 100
    pack append . .view {left fillx filly expand} .buttons {left fill}

    frame .buttons.grid1
    frame .buttons.grid2
    frame .buttons.grid3
    frame .buttons.grid4
    frame .buttons.grid5
    pack append .buttons \
.buttons.grid1 {top fillx} \
.buttons.grid2 {top fillx} \
.buttons.grid3 {top fillx} \
.buttons.grid4 {top fillx} \
.buttons.grid5 {top fill expand}

    button .buttons.grid1.bx -text "<-" -command "event 1" -height 3 -width 5
    button .buttons.grid2.by -text "<-" -command "event 2" -height 3 -width 5
    button .buttons.grid3.bz -text "<-" -command "event 3" -height 3 -width 5
    button .buttons.grid4.bw -text "<-" -command "event 4" -height 3 -width 5
    button .buttons.grid1.b1 -text "1" -command "type 1" -height 3 -width 5
    button .buttons.grid1.b2 -text "2" -command "type 2" -height 3 -width 5
    button .buttons.grid1.b3 -text "3" -command "type 3" -height 3 -width 5
    button .buttons.grid2.b4 -text "4" -command "type 4" -height 3 -width 5
    button .buttons.grid2.b5 -text "5" -command "type 5" -height 3 -width 5
    button .buttons.grid2.b6 -text "6" -command "type 6" -height 3 -width 5
    button .buttons.grid3.b7 -text "7" -command "type 7" -height 3 -width 5
    button .buttons.grid3.b8 -text "8" -command "type 8" -height 3 -width 5
    button .buttons.grid3.b9 -text "9" -command "type 9" -height 3 -width 5
    button .buttons.grid4.bc -text "Cont" -command "event c" -height 3 -width 5
    button .buttons.grid4.b0 -text "0" -command "type 0" -height 3 -width 5
    button .buttons.grid4.be -text "Enter" -command 'enter" -height 3 -width 5

    pack append .buttons.grid1 \
        .buttons.grid1.bx {left padx 20 pady 15 expand} \
        .buttons.grid1.b1 {left padx 20 pady 15 expand} \
        .buttons.grid1.b2 {left padx 20 pady 15 expand} \
        .buttons.grid1.b3 {left padx 20 pady 15 expand}

    pack append .buttons.grid2 \
        .buttons.grid2.by {left padx 20 pady 15 expand} \
        .buttons.grid2.b4 {left padx 20 pady 15 expand} \
```

```
        .buttons.grid2.b5 {left padx 20 pady 15 expand} \
        .buttons.grid2.b6 {left padx 20 pady 15 expand}

    pack append .buttons.grid3 \
        .buttons.grid3.b7 {left padx 20 pady 15 expand} \
        .buttons.grid3.b7 {left padx 20 pady 15 expand} \
        .buttons.grid3.b8 {left padx 20 pady 15 expand} \
        .buttons.grid3.b9 {left padx 20 pady 15 expand}

    pack append .buttons.grid4 \
        .buttons.grid4.bw {left padx 20 pady 15 expand} \
        .buttons.grid4.bc {left padx 20 pady 15 expand} \
        .buttons.grid4.b0 {left padx 20 pady 15 expand} \
        .buttons.grid4.be {left padx 20 pady 15 expand}

    button .buttons.grid5.state -text "Statement"   -width 30 -command "event x"
    button .buttons.grid5.cash  -text "Money"    -width 30 -command "event x"
    button .buttons.grid5.card  -text "Card In/Out"    -width 30 -command "event x"
    button .buttons.grid5.quit  -text "Quit"     -width 30 -command "destroy ."

    pack append .buttons.grid5 \
.buttons.grid5.quit  {bottom padx 80 pady 20} \
.buttons.grid5.state {bottom padx 80 pady 20} \
.buttons.grid5.cash  {bottom padx 80 pady 20} \
                .buttons.grid5.card  {bottom padx 80 pady 20}

    text .view.text1 -border 0 -height 20 -width 50 -state disabled\
-font "-*-helvetica-bold-r-normal--*-180-*-*-*-* iso8859-1"
# -justify right -text "a text"

    pack append .view .view.text1 {fillx filly}

}

proc type {a args} {
    global buffer
    append buffer $a
    .view.text1 configure -state normal
    .view.text1 insert end $a
    .view.text1 configure -state disabled
}

proc enter {} {
    global buffer

    event $buffer
    set buffer ""
}

#proc event {args} {
#    puts stdout $args
#}

#  "public" procedures...

proc clear {} {
    .view.text1 configure -state normal
    .view.text1 delete 1.0 end
    .view.text1 configure -state disabled
}

proc out {a args} {
    .view.text1 configure -state normal
    .view.text1 insert end "\n"
    .view.text1 insert end $a
    .view.text1 insert end "\n"
    .view.text1 configure -state disabled
```

```
}

proc lout {a b c d args} {
    .view.text1 configure -state normal
    .view.text1 delete 1.0 end
    .view.text1 insert end "Select:\n"
    .view.text1 insert end "                                                "
    .view.text1 insert end $a
    .view.text1 insert end "\n\n\n"
    .view.text1 insert end "                                                "
    .view.text1 insert end $b
    .view.text1 insert end "\n\n\n"
    .view.text1 insert end "                                                "
    .view.text1 insert end $c
    .view.text1 insert end "\n\n\n"
    .view.text1 insert end "                                                "
    .view.text1 insert end $d
    .view.text1 insert end "\n"
    .view.text1 configure -state disabled
}


mainwindow

spawnchannels ./tank $argv
#  end
```

## C.8 ATM — Haskell Code

```
module Main where

{-
-- The "bank" program...  A simple auto-teller.
--
-- We're going to use continuations for this one.
--
-- Plus a Tcl/Tk front-end!
--
-}

main = readChan epipe exit          $
       \input -> atm (lines input)

-- The behaviour of an atm specified in continuations...

atm :: Result
atm = insert_card_message           $
      card_in                       $
      enter_pin_message             $
      customer

customer :: Result
customer =
      pin_no                        $
      \pin -> validate_pin_no pin   $
      \valid_pin -> case valid_pin of
       Pin_OK -> service_prompt       $ services
       Retry  -> retry_message        $ customer
       Thief  -> keep_card_message    $ atm
       Wally  -> learn_number_message $ eject_card $ atm
       _      -> mydone

services :: Result
services =
      service                       $
      \which -> case which of
       Request_Cash       -> cash
       Request_Check_Book -> acknowledge_check_book      $ more
       Request_Balance    -> show_balance                $ more
       Request_Statement  -> print_and_proffer_statement $ more
       _                  -> mydone

cash :: Result
cash =
      amount_prompt                 $
      amount                        $
      \amount_query -> case amount_query of
       Amount_Hopeful -> sorry_but_message $ cash
       Amount_OK      -> confirm_prompt    $
                         confirm           $
                         \doit -> case doit of
                          Confirm -> proffer_card $ take_card $
                                     proffer_cash $ take_cash $ atm
                          Cancel  -> more
                          _       -> mydone
       _              -> mydone

more :: Result
more =
      eject_card                    $
      another_service_message       $
      eject                         $
```

```
      \answer -> case answer of
       Continue -> service_prompt  $ services
       Card_Out -> atm
       _        -> mydone

--

-- messages...

insert_card_message  = nmessage "Please insert your card for service..."

enter_pin_message    = nmessage "Please type your PIN."

keep_card_message    = message "Sorry, too many tries, I'm keeping it!"

learn_number_message = message "Sorry, wrong number!"

retry_message        = message "Incorrect PIN, please try again."

service_prompt       = messages "Cash" "Check Book" "Balance" "Statement"

acknowledge_check_book =ack "A cheque-book will be sent out to you.\nPress Cont"

show_balance           = ack "Your balance is <some-amount>.\nPress Cont"

print_and_proffer_statement = ack "Please take your statement."

amount_prompt     = message "Please type an amount of cash."

sorry_but_message = nmessage "You'll be lucky!"

confirm_prompt    = messages "Continue" "Cancel" "" ""

proffer_card      = nmessage "Please take your card."

proffer_cash      = nmessage "Please take your money, have a nice day!"

eject_card        = message "here's yer card."

another_service_message = messages "Finish Now" "Further Service" "" ""

--

-- hit return...

card_in = hit_return

take_card = hit_return

take_cash = hit_return

-- functions for the "Reply" types...

data Valid_Replys = Pin_OK | Retry | Thief | Wally
data Service_Replys = Request_Cash
                    | Request_Check_Book
                    | Request_Balance
                    | Request_Statement
data Amount_Replys = Amount_Hopeful | Amount_OK
data Confirm_Replys = Confirm | Cancel
data Eject_Replys = Continue | Card_Out

int2valid 1 = Pin_OK
int2valid 2 = Retry
int2valid 3 = Thief
```

```
int2valid 4 = Wally

int2service 1 = Request_Cash
int2service 2 = Request_Check_Book
int2service 3 = Request_Balance
int2service 4 = Request_Statement

int2amount n | n < 30     = Amount_OK
             | otherwise = Amount_Hopeful

int2confirm 1 = Confirm
int2confirm 2 = Cancel

int2eject 2 = Continue
int2eject 1 = Card_Out

-- functions to take user input into various types...

validate_pin_no :: Int -> (Valid_Replys -> Result) -> Result
validate_pin_no = continuation int2valid

pin_no  = number_input id

service = number_input int2service

amount  = number_input int2amount

confirm = number_input int2confirm

eject   = number_input int2eject


continuation :: (b -> a) -> b -> (a -> Result) -> Result
continuation x y z = z (x y)

noop x = x

-- difficult stuff...

type Result = [String] -> [Response] -> [Request]

type NumCont = Int -> Result

ack :: String -> Result -> Result
ack mess cont =
    \input -> appendChan cpipe ("out \""++mess++"\"\n") exit $
      case input of
        (1:ls) -> cont ls
        []     -> done

nmessage :: String -> Result -> Result
nmessage mess xx =
    \input -> appendChan cpipe ("clear ; out \""++mess++"\"\n") exit (xx input)

message :: String -> Result -> Result
message mess xx =
    \input -> appendChan cpipe ("out \""++mess++"\"\n") exit (xx input)

messages :: String -> String -> String -> String -> Result -> Result
messages mx my mz mw xx =
    \input -> appendChan cpipe
              ("lout \""++mx++"\" \""++my++"\" \""++mz++"\" \""++mw++"\"\n")
              exit (xx input)

mydone input = done

hit_return :: Result -> Result
```

```
hit_return cont =
          \input -> case input of
            (_:ls) -> cont ls
            []     -> done


number_input :: (Int -> a) -> (a -> Result) -> Result
number_input f cont =
          \input -> case input of
            (_:ls) -> cont (f (read l)) ls
            []     -> done

--

cpipe = "/dev/fd13"
epipe = "/dev/fd14"
rpipe = "/dev/fd15"
```