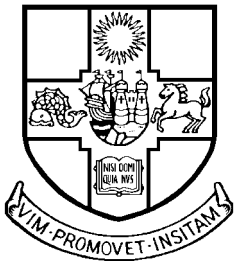
The Brisk Project: Concurrent and Distributed Functional Systems

Ian Holyer Neil Davies Chris Dornan

June 1995 **CSTR-95-015**



University of Bristol

Department of Computer Science

Also issued as ACRC-95:CS-015

The Brisk Project:
Concurrent and Distributed Functional Systems

Ian Holyer Neil Davies Chris Dornan

Department of Computer Science, University of Bristol, UK

Abstract

The Brisk project has been set up to investigate the possibility of extending the expressive power of pure functional languages. The aim is to be able to build concurrent and distributed working environments completely functionally, reversing the usual situation in which functional programs are regarded as guests within a procedural environment. This paper gives an overview of the project.

The Bristol Haskell System, or Brisk for short, is based on a compiler for the Haskell functional programming language which is used to provide practical support for this research, and to demonstrate its results. The compiler adds a purely deterministic form of concurrency to Haskell in order to improve support for interactive and distributed programming. This has been used, for example, to build an interface to the X window system. Features have also been added to support the dynamic loading and migration of code. This allows for a purely functional implementation of long-lived shell programs which manage files, processes and communications.

1 Introduction

Current functional language implementations suffer from having a guest status within procedural operating systems. Typically, each program runs as a single process and accesses operating system facilities via sequential input/output interfaces. This is restrictive and unnatural from a functional point of view. It is difficult to implement reactive systems such as graphical user interfaces, and it is impossible to implement larger scale systems programs which often involve unavoidable concurrency.

The Brisk project was set up to solve this problem by providing a deterministic form of concurrency to increase the expressive power of pure functional languages, without losing any of their pure and mathematically simple properties. Brisk's concurrency allows reactive graphics programs to be implemented cleanly and conveniently. Efficiency is also reasonable, so that it is possible to implement interactive graphics games for example. Other features such as the dynamic loading of code are added to increase expressive power in other ways, allowing development environments and distributed systems to be designed in a purely declarative and deterministic way.

This paper gives a broad overview of the Brisk project, describing a number of topics in brief outline. We hope that some of the topics will be expanded in more detail in later papers.

The Brisk system provides a compiler which implements deterministic concurrency. This compiler is described briefly in Section 2. Concurrency allows interfaces to operating system facilities to be made more modular, so that independent facilities are accessed via separate threads of execution, as described in Section 3. In particular, a natural and convenient interface to window systems can be provided, in which programs can be written in a reactive style – see Section 4. Further extensions to Haskell allow modules containing new functions and types to be loaded dynamically into an already running program. In Section 5, we show how these extensions allow a development environment for Brisk to be implemented entirely in Brisk.

In the remaining sections, we discuss the design issues involved in making extended applications deterministic, and thus implementable using Brisk. The design techniques are of interest in their own right, independent of functional programming, as they change the look and feel of concurrent applications, making them more predictable, repeatable and even provable. In Section 6 we describe the issues of filestore and process management which would be involved in building a multi-process working environment for a single user. Distributed applications are described in Section 7, together with the questions of communication and global identification of resources which they raise. Finally, in Section 8, we discuss some of the issues raised by multi-user applications.

2 The Brisk Compiler

The Brisk compiler is a conventional compiler for the Haskell language [4]. For portability, the compiler translates Haskell modules into C, and a C compiler then completes the translation into machine code.

One innovation is the addition of deterministic concurrency, as described by Holyer & Carter [1]. Concurrency is introduced into Brisk not to speed up programs using parallel processors, but rather to make Haskell more expressive so that programs can be written with concurrent behaviours which cannot be achieved with a normal sequential version of Haskell. Also, the initial aim in Brisk is to study concurrent programming in a shared memory setting to avoid the restrictions and complications involved in using distributed communicating processes.

Thus, in the initial Brisk system at least, a compiled concurrent program still runs as a single operating system process. The run time system supports lightweight threads of execution which share common global data in a single heap. This is sufficient to examine all the applications up to Section 6. When distributed systems are discussed in Section 7, communication issues will be re-examined.

The Brisk run time system uses the well-known techniques developed for simulating parallelism using lightweight threads, as described by Peyton Jones & Lester [5] for example, and as described by Peyton Jones & Finne [8] for use in Concurrent Haskell. These provide simple and efficient thread switching and data locking in the context of graph reduction. In addition, the Brisk run time system provides centralised I/O control so that a thread which suspends on I/O does not prevent other threads from continuing.

Although conventional implementations of Haskell are sequential, the language itself is not inherently sequential. Thus the Brisk compiler needs no extension to the standard

Haskell syntax in order to provide concurrency. The only change is that the enhanced run time system makes it possible to provide extra library modules which provide concurrent interfaces to external facilities such as window systems.

The denotational semantics of Haskell also needs no extension; the conventional lazy value semantics is sufficient to describe deterministic concurrency. The operational semantics can be described in terms of independent demands on a number of expressions within the program. Typically, the demands come from output streams or action sequences. Other extensions to Haskell mentioned in later sections also have a minimal effect on syntax or semantics.

3 Concurrent Interfaces

Input and output in version 1.2 of Haskell, the version described in the original Haskell report [4], is carried out via streams of requests and responses, or by using functions with continuations. Since then a monadic style has been developed for I/O. This style provides a uniform approach to the problem of interfacing with procedural or other state-based services, and increases the convenience and composability of procedural actions. Interfacing in Brisk will be described in monadic terms, using the conventions currently being proposed for version 1.3 of Haskell [10], although at the time of writing the details of Haskell 1.3 have not yet been standardised.

With the monadic interfacing technique, a number of I/O actions are provided. These actions can be thought of as functional versions of I/O procedures provided by the operating system. Actions can be regarded as functions which transform the current I/O state of the program. However, actions are treated as an abstract type so that the programmer does not have direct access to the program state. This restriction prevents the programmer from creating multiple versions of the state. Instead, basic actions are combined into a single linear sequence, using combining operators such as `>>` and `>>=` provided in Haskell 1.3.

Launchbury & Peyton Jones [6], [7] describe extensions of monadic interfacing to other services besides I/O. All external services happen via actions on a specific `RealWorld` state type representing the external system state. This provides I/O and general C language procedure call services, but forces all C procedure calls to happen in a single linear sequence.

In Brisk, using concurrency, access to external services can be made less restrictive. A number of separate linear sequences of procedure calls can be executed concurrently, provided that the external effects produced by different sequences are independent of each other. The sequences are executed independently, each at its own rate, except for the usual synchronising effect of any data dependencies within the program. In addition, different sequences can have different state types, providing the opportunity for a more modular approach to interfacing with external services.

Launchbury & Peyton Jones [7] describe generic `runST` and `interleaveST` functions which perform a given compound action either on a given initial state or on a shared state, and Peyton Jones & Finne describe a generic `forkIO` function for carrying out separate compound I/O actions concurrently. These generic facilities lead to safety problems, and make it impossible to exclude non-deterministic behaviour.

In the Brisk approach any number of independent state-based services can be provided

in separate modules, each with its own state type. There are no generic facilities for starting or forking actions. Instead, specific versions of them are provided, each of which is guaranteed to be safe and deterministic. For example, a version `runArray` of the generic `runST` function can be provided as part of an array package; in general, versions of `runST` are provided for internal services, but not for I/O or other external services.

Similarly, safe forking primitives are provided. One example is an action to read the contents of a file concurrently with the main I/O thread, the file becoming inaccessible from the main thread. In fact the `hGetContents` action proposed for Haskell 1.3 splits the state in this way, though concurrency is not involved. Other examples are an I/O action which starts up an X Window session independently of the main I/O sequence, or an X Window action which creates a new independent subwindow. These can be guaranteed to be safe provided the graphics facilities available on any one window do not affect other windows or subwindows.

In general, a forking primitive takes a complete subaction as an argument and splits off a part of the main state for that subaction to act on. The new subaction may or may not need another thread of execution to handle it, depending on whether or not it represents a new source of demand to the program as a whole, or whether it is driven by data dependency on its result. The part of the state that is split off is inaccessible from the main sequence. Joining the split state back together can also be achieved, though that is not discussed here.

A Brisk program can thus have many separate action sequences, each driven by its own independent demand, and each capable of cooperating independently with external services. This implies a need for concurrent I/O; if one thread is suspended while waiting for input, for example, the other threads must be allowed to continue. In practice, one can think in terms of a central I/O controller which keeps track of all input channels. If a number of threads are waiting for input on separate input channels, the controller must be able to respond to the first input item to arrive, regardless of which channel it arrives on. This amounts to a non-deterministic timed merge. However, all the controller does is to wake up the relevant thread, so the program as a whole is not able to produce different results depending on the relative times at which items arrive on the input channels. The timings of the inputs affect only the timings of the outputs, as with sequential programs.

This deterministic form of concurrency is a compromise. The separate threads of execution and the concurrent control of I/O provide a level of expressiveness which goes beyond that of sequential implementations. On the other hand, the effects are purely deterministic which means that Brisk is less expressive than systems which allow full non-determinism.

This provides an interesting challenge. The precise effects of many conventional systems programs and reactive programs cannot be obtained directly in Brisk, because the effects themselves are inherently non-deterministic. However, it is often possible to redesign such programs so that they provide essentially the same services, but using only purely deterministic effects. The remaining sections of this paper discuss several common situations in systems programming, and demonstrate that the non-deterministic effects used in conventional approaches are usually unnecessary, and indeed undesirable.

4 A Window System

Conventional window systems such as X offer graphics facilities on computer screens organised in terms of a hierarchical collection of windows. Windows come in all sizes from the root window representing the whole of the screen down to individual buttons, sliders and menu items. Programs written using window systems often have complex reactive input and output behaviour.

Previous functional interfaces to window systems have either involved non-determinism or else been difficult to program. In the latter case, there is a need to deterministically merge separate graphics streams into a single stream to send to the screen. This causes "plumbing" problems which force the structure of window programs to match the structure of their windows. As a matter of fact, these problems occur with window interfaces in sequential procedural languages; this is not just a functional programming problem. The difficulties are well described by Noble & Runciman [11].

The programming model conventionally presented by window systems is one in which separate windows are almost completely independent of one another. Graphics which are drawn in a window do not affect the parent window or any other window obscured behind it. Moving a window exposes graphics previously drawn on the window behind, even if they were drawn while the back window was obscured.

We capitalise on this by providing primitives which guarantee that windows are completely independent. The interface to the window system can then be made concurrent, with each window having a separate output thread for the the graphics which appear in it. Each thread can perform its own output, with no need for merging. Such an interface is clean and simple, and yet captures most of the useful patterns of behaviour which window programs typically use. In addition, such a window system provides a rich source of concurrency with which to experiment further with concurrent program design.

An initial version of a concurrent window interface of this kind, called BriX, has been designed and implemented by Serrarens [3]. There are several issues involved in making such an interface powerful, and yet deterministic. At present, the interface is at the lowest (Xlib) level, since the higher level libraries often used with the X window system are designed using models of behaviour which are difficult to capture functionally.

The Xlib procedure calls can be classified into various different kinds. First, some can be described as *forking actions*. The main examples of this are:

```
xOpenDisplay :: String -> X () -> IO ()

xCreateSimpleWindow :: Position -> ... -> X () -> X ()
```

The `xOpenDisplay` function is called as part of a program's main I/O sequence. It takes the name of a display, and a compound X action representing the processing to be performed on the program's main window. Then it starts up a new thread to perform this processing separately from the main I/O thread. The `xCreateSimpleWindow` function is called as an action on a window which creates a new subwindow. The last argument is the compound action to be performed independently on the subwindow.

Second, many procedures can be classified as *graphics actions*. These write or draw or paint on a window's canvas. This is the simplest kind of procedure, modelled as an action

on the current window (which is incorporated into the state associated with the action type `X a`). For example:

```
xDrawString :: Position -> String -> X ()
```

Third, some procedures are *control actions* which involve changing the relationship between a window and its surroundings, eg moving or raising a window. In Xlib, such a procedure would be called with the window to be moved or raised as an argument. However, this can lead to non-determinism; if two overlapping windows attempt to raise themselves above each other, the result depends on the relative timing of the two operations. To resolve this, such operations are described instead as actions by the parent which take a subwindow identifier as an argument. Since these operations do not interact with the graphics inside the subwindows, and only affect the relationship of subwindows with each other rather than with unrelated windows, this approach retains the deterministic independence of windows.

In low level libraries such as Xlib, all input events such as key presses or mouse clicks are sent to a program in a single stream. The functional model of events is that they are received by the main window, and then appropriately filtered and passed on to subwindows. This ensures that the thread associated with a window has access to information about relative timings of events in the subwindows where necessary. Conventional dispatch mechanisms within X toolkits which send events directly to the windows that require them can be regarded as an optimisation in cases where parent windows have no interest in their children's events.

Some events are generated by the window system itself. These are usually caused directly by user operations such as moving windows on the screen, and so the window system can insert them at correctly synchronised points into the main event stream. However, some such events, such as exposure events for example, can lead to unfortunate effects. The intention of exposure events is that the program should respond by redrawing windows which have been uncovered. However, the program is free to draw something different from the graphics which were originally sent, violating the independence of windows. Since exposure events are just an efficiency mechanism to save the window system from having to store an arbitrary amount of covered-up graphics, they represent an unnecessary complication for the programmer. With our functional approach, it is sensible to have one or more built-in redrawing techniques so that redrawing is not the responsibility of the programmer.

There are situations in which non-determinism seems indispensible. For example, suppose a window displays a number of images in sequence, and you want to be able to click on a button to stop the program at the one you want. The problem is that of determining the relative timings of the moment at which the next image is displayed and the moment at which the button is pressed. One way to solve such problems is to allow synchronisation requests to be embedded in a graphics stream. The window system responds by returning synchronisation events in the event stream. The window system can ensure that the moment at which the synchronisation event is returned is definitely after all previous graphics have appeared on the screen and definitely before any further graphics appear. The position of the button click event relative to the synchronisation events can be used by the program to determine how to react. Thus this timing problem is delegated to the window system itself, which is best able to achieve accurate synchronisation.

It is hoped that this functional window interface will provide a convenient programming environment, in which the structure of a program can be separated from the structure of its windows.

5 A Development Environment

Consider the problem of constructing a development environment for compiling, debugging and running functional programs. We want to be able to regard the development system as a single, long-running program, implemented in a functional language and providing a functional 'look and feel' to the user.

A naive view of a compiler is as a function from the source text to a value. A simple compiler for numerical functions might have a declared type and might be used as:

```
compile :: String -> (Int -> Int)

square = compile "square n = n*n"
```

The program implementing the development environment would accumulate extra, dynamically determined function values by compilation as time progressed. There are several problems with this naive view. The first is that compilation does not happen in isolation, but rather in the context of previously compiled modules or functions. Another is that the type of the value produced is variable rather than fixed. Also, source programs may define new data types, and it is not clear how the development environment program can accumulate extra, dynamically determined types.

It is possible to solve these problems in a reasonably simple way by extending the functional language a little. First, compilation can be regarded as an I/O operation rather than a stand-alone function. Modules can then be compiled in the context of a module store which holds a self-consistent collection of successfully compiled modules. Second, the run-time system of the functional language needs to be able to support dynamic loading and linking of compiled modules. Finally, it is possible to introduce an extra type into the language, called `Dynamic` for example, which represents the ('tagged') union of all data types, including newly compiled ones. Values can be extracted from this type by a form of pattern matching which matches values of type `Dynamic` against type constructors. Thus a dynamic value can be extracted and used provided it is of a type which is known in the current context; if not, then the value can still be passed around without looking inside it.

There are both semantic and practical details to be sorted out here, and these are under investigation, but the result is a further increase in the expressiveness of the language.

6 Process Management

Extending the ideas in the last section, it is possible to develop a purely functional multi-process environment within which a single user can run programs. We want to be able to regard the entire system, including all the separate processes which are run and the support provided for them, as a single long-running functional program.

The dynamic loading and linking described briefly in the previous section can be used to start up separate processes. In addition, there are new issues of file management and process management which arise.

If two independent processes are allowed to perform arbitrary input and output operations on the file store, then non-deterministic effects can result. The state of the file store can depend on essentially arbitrary relative timings, eg of two processes writing to the same file. Indeed, this is what happens in conventional operating system environments, and such non-deterministic effects form some of the major sources of surprises and pitfalls for users.

To avoid this, it is necessary to be very clear about ownership. Suppose that all processes are started up from a manager or 'shell' process which owns the directory structure of the file store and has built-in facilities for altering it, but does not touch file contents. When a process is started up, it is called as a self-contained function. It is given the contents of the files it requires as arguments, and the results are put back into the file store by the manager as new file contents; it does not have any direct access to the directory structure. This division of responsibilities ensures that the state of the file store is deterministic, and that the manager can continue reliably regardless of the success or failure of the processes which it spawns. A unique way of identifying files is needed so as to avoid aliasing problems.

As some processes started up by the manager are long-lived, and the manager must continue while they run, the act of spawning a process is regarded by the manager as an action which replaces the contents of the result files immediately by unevaluated expressions. This corresponds to the usual state of affairs in conventional systems where files are often in the process of being written to. Also, separate threads of execution are created to evaluate the contents of the result files; this amounts to 'speculative parallelism' in the sense that file contents are evaluated without knowing whether or when the file contents are going to be used. The advantage is that this creates the same level of persistence as in conventional systems, where a system 'crash' need not do too much damage.

A mechanism is needed to deal with long-lived programs such as editors which have facilities for dynamically choosing new files to work on. One way of achieving this is by having the editor program take a stream of files as an argument. Editing a new file then involves two commands; one to tell the manager to send a particular new file, and one to the editor to tell it to read the contents in from its stream argument. A 'drag-and-drop' graphical operation could conveniently combine these two coordinated commands into a single user action.

An interesting problem arises with the need to abort runaway processes. In conventional systems, this is done by sending a signal to the errant process itself. This is not acceptable here because it would require a non-deterministic merge of such signals with the normal inputs of the process, and anyway the process might not be listening for such signals. Instead, a command can be given to the manager process. If the manager keeps old versions of files, the result files from the process in question can be made to revert to their old contents, so that no references to the new contents remain. The runaway process then has no sources of demand driving it, so it stops. There are also no references to the space it occupies, so the space can be garbage collected.

There are many other details which need attention. One important one is to find ways

to run existing procedural programs as guest processes, with safe wrappers around them so that they can be given functional descriptions. This avoids the need to rewrite large quantities of system software. The examples provided here illustrate the fact that more is possible using purely deterministic effects than at first meets the eye, and the result could be a system which users may find preferable to the present anarchic state of affairs.

7 Distributed Systems

All the above examples can be imagined as single long-running functional program in which separate threads or processes share a single large heap. In general, however, concurrency is associated with multiple processes with separate memories which communicate with each other. Indeed, this was the way in which we first approached concurrency, see Carter [2]. Programs can be distributed either over closely coupled processors in a parallel computer, or over loosely coupled computers in a network; many of the issues are similar in both cases. Some of the issues associated with turning Brisk into a distributed system have been investigated, and a few of these ideas are presented here.

Although distributed systems are likely to be multi-user, we defer the issues which this raises to the next section. This means that a distributed program can be regarded as being equivalent to a single-heap program. This equivalence requires that all details concerning communication protocols etc should be hidden below the functional level.

There are great theoretical advantages to be had from this equivalence. A collection of communicating processes in which the local inter-communications are hidden is completely equivalent to a single process of the same kind; this simple composability is in sharp contrast to most procedural models of communicating processes. The semantics and behaviour of a program can be studied independently of the its distributed nature, and indeed it may be possible to allow the functional design of programs to be separated from the design of their distribution. This is again in marked contrast to procedural systems, eg those using remote procedure call or client-server models.

The main problem in practice is the restrictive nature of typical communication facilities. It is possible to use communications to implement a single distributed, heap. Unfortunately, doing this in a naive way is likely to be too inefficient.

In particular, it is unreasonable to expect demand to be transmitted in the system in the same way as it would be in a single-heap version of the program. Rather, each communication channel can represent an independent source of demand on its supplier process. This can work well, allowing for conventional buffering of channels etc, provided that the communication protocols used have no semantic effect at the functional level. They just correspond to the addition of some speculative parallelism.

It is important to ensure that the restrictions imposed by communication channels for efficiency's sake do not unduly restrict what the programmer can achieve. In particular, we want to allow for the mobility of code as well as data. It is easy enough to arrange for code to be shipped across communication channels, but there are issues of version handling and portability to take care of. For version handling, it is important that programs or pieces of code are known by globally unique identifiers, with every version getting its own identifier, rather than just being known by name. The easiest way to ensure portability is to keep

code in interpretable form, though other schemes such as local re-compilation are possible.

Other issues which arise are decisions as to whether to transmit data in evaluated or unevaluated form, and whether to copy or donate values, how to arrange for distributed garbage collection, and so on. However, there are various benefits to be had from making such a scheme work. For example, given the discussions in previous sections, it is possible to run code obtained from elsewhere in a safe side-effect free manner so that, for example, viruses are not possible.

8 Multi-user Systems

Systems with multiple users raise further issues. In fact, in a multi-user system, it is no longer possible to avoid non-determinism completely. When two people perform update operations on shared data, there must be some way to determine in what order to carry out the operations.

However, the expressiveness of the systems described so far make it possible to confine non-determinism to a few well-understood places in the system, and to use techniques such as time-stamps which would not be practical if they pervaded the system.

The minimum level of cooperation between users is the ability to view each other's files. One way to achieve this which fits in with the idea of keeping old versions of files is that each user sees a fixed snapshot of another user's file store, at the moment when the second user last logged out. This requires only the timed merge of login and logout requests, and ensures that a system administrator can update publicly accessible files safely, in the knowledge that only when the files have been checked for consistency and the administrator has logged out will anyone see the changes.

Closer cooperation, such as access to a shared database, requires time-stamped transactions on a smaller scale. Provided that it is obvious to the user when timed transactions are being carried out, eg by having a special kind of window in which it is clear that times are being attached to requests, this can work well.

9 Conclusions

The aim of the Brisk project is to demonstrate that a deterministic form of concurrency, while not as expressive as non-determinism, allows for a great variety of effects to be achieved. Indeed many, if not most, systems programs and reactive systems can be re-designed to use such concurrency without any significant loss of functionality.

The discussions in this paper, although brief and incomplete, already demonstrate that much more is possible than one might expect at first sight. There are many theoretical advantages to determinism; functional languages remain referentially transparent making both formal and informal reasoning about programs and their I/O behaviour much easier. It has practical advantages too; it encourages people to design complex systems in a way which is predictable and repeatable for their users. These design principles may have applications beyond functional programming.

References

[1] Holyer & Carter, *Deterministic Concurrency*, Functional Programming Glasgow 1993, Workshops in Computing Series, Springer-Verlag,
(available from `http://www.cs.bris.ac.uk`).

[2] Dave Carter, *Deterministic Concurrency*, submitted as MSc thesis, Computer Science Department, Bristol 1994, (available from `http://www.cs.bris.ac.uk`).

[3] Pascal Serrarens, *BriX – A Deterministic Concurrent Functional X Windows System*, technical report, Bristol 1995, (available from `http://www.cs.bris.ac.uk`).

[4] SIGPLAN Notices, Vol. 27, No. 5, *Haskell Special Issue*, ACM 1992

[5] Peyton Jones & Lester, *Implementing Functional Languages*, Prentice Hall 1992.

[6] Launchbury & Peyton Jones, *Lazy Functional State Threads*, Proceedings Programming Language Design and Implementation, Orlando 1994 (and Glasgow technical report).

[7] Launchbury & Peyton Jones, *State in Haskell* 1995 (LASC, to appear).

[8] Peyton Jones & Finne, *Concurrent Haskell*, 1995 (to appear).

[9] Finne & Peyton Jones, *Composing Haggis*, 1995 (to appear).

[10] Kevin Hammond (ed), *Report on the Programming Language Haskell (version 1.3)*, Glasgow, available via `ftp` from `ftp.dcs.glasgow.ac.uk`.

[11] R. Noble & C. Runciman, *Functional Languages and Graphical User Interfaces – a review and a case study*, technical report YCS 223, Department of Computer Science, University of York, 1994