Feris: A Functional Environment for Retargetable Interactive Systems

Robert Ennals

King's College, Cambridge University,
rje33@cam.ac.uk

Abstract. Feris is a library and runtime environment for creating interactive programs. Programs written using Feris may be run as GUI applications, but also in other ways, such as as web pages, text mode applications, or embedded in another program.
In addition to being of practical use in itself, Feris demonstrates a programming model based on translation between different abstractions, and an approach to interfacing functional languages with the outside world. The core of the system is a network of translators that translate between different abstractions. Applications can describe their interface in terms of whatever abstractions they like (e.g. diagrams or actions), and low level environments can describe their interface using another set of abstractions (e.g. pixels or text), and translators to translate between them.
Also noteworthy is that Feris programs cannot directly perform actions on global state, even when working with files.

1 Introduction

It is often useful for computer programs to have some way of interacting with a user. Usually this is done using a Graphical User Interface (GUI). However there are many other ways in which a program can interact with a user, including speech driven systems, command line programs, web pages and embedding inside another program. Additionally, there are many ways in which the same program could be presented in any of these environments. For example, one might want to have different types of GUIs for experts, beginners, or the partially sighted.

A typical GUI based application will hold inside it some data that is being worked on. For example it might be an equation editor that has stored inside it some representation of an equation. In order to talk to an external GUI environment, this data needs to be translated into low level data such as pixels and basic GUI widgets.

The popular convention has been to do this translation inside the application. For example the app may apply a function to its high level equation description to obtain a bitmap that represents it. The disadvantage of this approach is that the only information available to the external environment is this low level information, and this is too target specific to be easily used for an alternative target such as a speech or web interface.

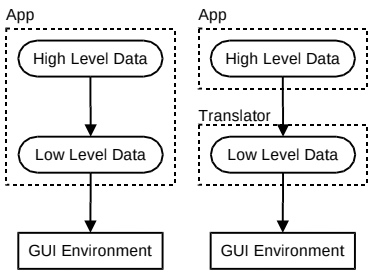

Fig. 1. Moving the translation from high level data to low level data outside the application

Feris takes an alternative approach. The application exports its high level data (which can be in any format) to the outside world. This is then translated into low level data by a set of translators that live outside the application and are managed by the environment. This difference is illustrated by figure 1. Moving this translation outside of the application allows the environment to take control of the process and handle low level targets that the application may not have been aware of.

Popular existing systems such as COM/ActiveX/OLE [4], GTK [17], and Java's Swing [19] go a little way in this direction, but are more coarse grained, more GUI specific, and give more control to the application instead of to the environment.

1.1 Contributions

The key contributions of this work are the following:

- A programming model based on layers of translators, translating between interactive objects. In this model translators, applications, and components are all the same thing.
- The ability to retarget a single application to many types of interface, including such things as GUIs, Speech, and The Web, without going through a lowest common denominator. (We don't translate GUIs to web pages, or web pages to GUIs. We translate applications to both.)
- The ability to extend the retargeting mechanism in a modular way. New targets, and ways of translating into targets can be implemented by third parties, with only the lowest level environment interfacing needing to be done outside the core Feris system.
- A monad of callbacks, allowing translators to work with the objects they are translating, without giving them the full destructive power of the IO monad.
- An alternative approach for interfacing with the outside world, without using foreign language calls or global effects.
- The production of a usable implementation of this system

Fig. 2. The Number Edit App as a GTK GUI

2 Why a Functional Language?

Feris has been implemented around a simple strict functional programming language called Oval, however it is planned to port it over to Haskell. To aid readability, and to avoid having to explain Oval in this paper, all source code given in this paper has been edited to be valid Haskell. The only changes required were minor syntactic issues.

The primary reason for using a functional language was Monads. Monads can be used to allow one to create translators that can have effects on objects they are translating, without being able to perform global effects. First class functions were also a major attraction.

3 A First Example

Figure 2 is a screenshot of a program called `numedit_app` running in the Feris GTK [17] environment. It has an integer variable, and allows this integer to be increased and decreased by pressing two buttons. The source code for this program is given below:

```
 1: data NumEdit = NumEdit (Var Int)

 2: numedit_ui :: Registry -> NumEdit -> CB Group
 3: numedit_ui _ (NumEdit num) = return $ Group[
 4:       dy $ Number num,
 5:       dy $ Action "+" $ update (\x -> x+1) num,
 6:       dy $ Action "-" $ update (\x -> x-1) num]

 7: numedit_app :: CB NumEdit
 8: numedit_app = do
 9:   num <- new 123
10:   return $ NumEdit num
```

Let's break down what this program is actually doing.

- Line **1** declares a new datatype called `NumEdit`. This is an abstract description of a number that the user is able to edit, and says nothing about how this editing should take place.
- Line **2** is declaring the translator function that is to produce a user interface for a `NumEdit`.

– Line **3** returns as the result a group containing a number and two actions. This group is a description of a UI that the environment should present to the user. Each member is a UI control that should be be displayed.
– Line **4** describes a number item in the group. `Number` constructs a UI element that continuously displays the value of the given `Var Int`. `dy` is used to convert each UI item into a dynamic type, so that UI controls of different types can be put into one list.
– Line **5** describes an action item in the group. `Action` constructs a button that is labelled by the given text, and reacts to being clicked on by performing the given action. In this case, the action is to update `num` by applying an increment function to it.
– Line **6** does essentially the same thing as line **5**, but creates a button to decrement the integer, rather than to increment it.
– Line **7** defines a simple top level procedure. Its body is a command of type `CB Group` that creates a numedit object for the environment to display.
– Line **9** creates a new editable number called `num`, with initial value of 123. This is the number that will be edited by the user.

This program makes use of the following externally defined types and functions:

```
dy :: a -> Dynamic
update :: (a -> a) -> Var a -> CB ()
new :: a -> CB (Var a)
data Number = Number (Var Int)
data Action = Action String (CB ())
data Group = Group [Dynamic]
```

4 Running Programs

A running program will generally consist of three components (see figure 3).

– The Application itself (e.g. `numedit_app`)
– A set of translators
– A target environment (e.g. GTK GUI)

The application, and the translators are all dynamically loadable objects, and can be loaded in arbitrary combinations, and used with arbitrary target environments.

The application is a program, such as `numedit_app`, that provides a description of something that it wishes to have displayed. The description provided by the application is very high level, and not specific to any particular target environment. It may well use data structures defined specifically for that particular application.

The target environment is a low level C program that has the job of working with external libraries and the operating system in order to allow interaction

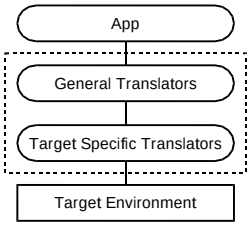

Fig. 3. Modules Involved in Running an App

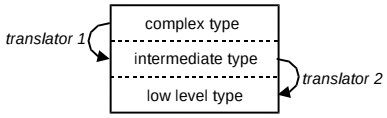

Fig. 4. The registry may apply several translators in series to do a complete translation.

with the outside world. The target environment needs to be given a very low level, target specific description of the interface it is presenting.

The application and the target use different sets of data structures to describe the user interface. It is thus necessary to translate between these two descriptions in order to allow them to interoperate. This is done by the of translators. In most cases, the translators are where the majority of the work will be done.

5 The Translator Registry

Translators convert between different types of data. Each translator has a source type and a destination type. Given an object of the source type, they can produce a corresponding object of the destination type. These two objects are linked such that any change made to one of these objects causes a corresponding change to be made to the other object (we ignore for the moment, the concept of one way translators). They thus behave like two views of one object.

Translations are managed by a translator registry. This registry has a table for each target it knows about (e.g. GTK, and Speech). This table maps each type it knows about to a function that can be applied to it to get it closer to the type that the target wants. A single translator will not necessarily take us all the way to our desired type. In the general case, one will need to apply the translation table several times in order to get something of the correct type for the target. This is illustrated in figure 4.

The current interface to the registry is given below. In order to translate an object from its current type, to the destination type for a target, one calls the `translate` function. In addition to the object to be translated, this function is also passed a registry from which to obtain the translation table and a

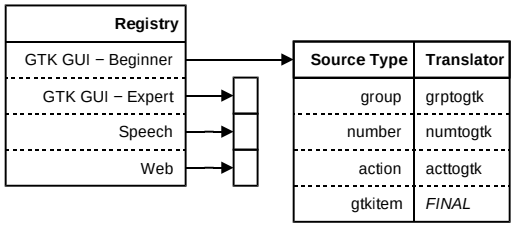

Fig. 5. A simple translator registry, with the GTK target expanded

string identifying the target that we are interested in. The `Registry` can contain translator tables for several different targets (see figure 5).

```
data Registry = ...
data Trans = Trans Dynamic | Final

translate :: Registry -> String -> Dynamic -> CB (Maybe Dynamic)
addtrans :: Registry -> String -> String -> Trans -> Registry
```

Strings are arguably an ugly way to identify translation targets. It is planned that a more elegant solution will be used in future.

6 Types

Some may consider it to be cause for concern that `translate` inputs and outputs objects of type `Dynamic`, and that type `Dynamic` is used to store objects of different types in `Group`s.

As far as we can see, static typing wouldn't bring any benefits here, as the success or failure of `translate` depends entirely on the contents of the `Registry`, and the `Registry` is, by its very nature, dynamic, and so not easily attacked with static typing.

Even if one created a type system that could check in advance whether applying a translator with a given Registry was guaranteed to succeed in all its translations, this would not buy very much, as the type checking would still have to be done at runtime, when the contents of the `Registry` was known.

One might similarly question why translators are managed as a set of functions conforming to a standard signature, rather than having them as implementations of a standard class operation. This is due to the desire to be able to manage several different translations between the same two types.

7 Editable Objects

At the foundation of the translator system is the `CB` monad. Translators may be written directly in terms of this monad, or in some form that can be translated into the `CB` monad.

The CB monad allows one to do several important things:

- Create an editable object (Var)
- Read an editable object
- Write to an editable object
- Cause a callback procedure to be executed whenever an object is written to
- Cause a procedure to be executed in the background

The types and operations involved in the CB monad are given below.

```
data CB a =  -- abstract
data Var a =  -- abstract
data Sched =  -- abstract

(>>=) :: CB a -> (a -> CB b) -> CB b
return :: a -> CB a
new :: a -> CB (Var a)
read :: Var a -> CB a
write :: Var a -> a -> CB ()

react :: (a -> CB Bool) -> Var a -> CB ()

later :: CB () -> CB ()
newsched :: CB () -> CB Sched
```

The abstract type CB a is a description of an operation that performs some imperative actions, and finishes with a result of type a. One can thus think of a monadic function of type a -> CB b, as being analogous to an imperative procedure with argument type a and result type b. As with the IO Monad [11], operations in the CB monad can only be executed by the outside environment, and not directly from within Oval evaluations.

The abstract type Var a is an editable variable with contents of type a. This is very similar to a BVar in FranTk [13], and to MutVar in the ST Monad [12]. The first five functions are all operations on these types, and are equivalent to similarly named functions in the ST Monad. **new** creates a new Var, and returns an identifier it. **read** obtains the current value of a Var. **write** stores a new value in a Var. **return** does nothing, and returns the given value. As with other monads, do notation can be used as a convenient shorthand for >>=.

react is a particularly important command. It allows one to request that a given procedure be executed when the given Var is written to. The callback procedure returns a boolean value that says whether it wishes to stay attached to the Var, and be executed in response to subsequent writes.

In order to make concurrent programming easier and avoid the need to worry about locking data structures, all callbacks are executed as atomic transactions using two phase locking [14].

newsched and **later** are used in special cases, and are discussed later.

We can now define some useful functions. Some of these were used in earlier examples, and some of these are used later.

```
update :: (a -> a) -> Var a -> CB ()
update f v = do{x <- read v; write v (f x)}
```

Update a `Var` by applying a function to its value.

```
reactalways :: CB () -> Var a -> CB ()
reactalways act v = react (\_ -> do{act; return True}) v
```

A variant on react, where we ignore the argument, and never cancel the callback.

8 Some Example Translators

A translator from type `a` to type `b` has type `Registry -> a -> CB b`. The `Registry` argument allows the translator to apply `translate` to any sub-objects within the object it is translating.

We have already given a translator from `numedit` into an abstract UI. We will now give as examples, the translators that translate the result of `numedit_ui` into a form that can be used by the GTK [17] target. The GTK target requires the GUI to be described in the form given below. For reference, the types used in the result of `numedit_ui` are repeated here also.

```
data GtkItem = GtkButton String (Var ()) | GtkText (Var String)
    | GtkHGroup [GtkItem] | ... -- further parts omitted

data Number = Number (Var Int)
data Action = Action String (CB ())
data Group = Group [Dynamic]
```

8.1 A Translator for Action

We will start out with the translator for `Action`. This needs to take an object of type `Action`, and produce a corresponding `GtkItem` object.

```
1: acttogtk :: Registry -> Action -> CB GtkItem
2: acttogtk reg (Action name proc) = do
3:       ubv <- new ()
4:       reactalways proc ubv
5:       return $ GtkButton name ubv
```

In this case, we are translating actions into GTK buttons. We could chose to translate actions into anything we liked, but buttons seem to be the most appropriate mapping. The button we produce has a label equal to the name of the action, and clicking on the button causes the action to execute its callback procedure.

The GTK Button constructor contains a string, and a `Var ()`. The string is the name of the button, and the `Var ()` is a variable that the button writes

to whenever it is pressed. Although the `Var` doesn't change value, writing to it causes any attached callbacks to be executed. One can thus cause a procedure to be executed whenever the button is pressed by making it `react` to the `Var`.

8.2 A Translator for Number

`Number` is slightly more complicated to translate. While our chosen set of GTK primitives doesn't contain a number editor, it does contain `GtkText`, a basic string editor.

The GTK widget displays the contents of the text string, and allows the user to edit the string. When the user edits the string, the new value is written to the `Var String`. When the value of the `Var String` changes, the GTK widget updates accordingly.

In order to use this widget to edit an `Var Int`, we need to translate changes in both directions. When the int changes, we want to update the string to contain the string representation of the integer. Likewise, when the string changes, we want to update the integer to contain the integer the string represents.

When doing these updates, we need to be careful to not cause circular updates. That is, if a change to the int causes a change to the string, we don't want this to cause another change to the int, which causes the process to repeat again. Fortunately the Feris standard library has a set of functions (implemented in Feris) that make avoiding such loops very easy. One of the most common of these functions is `link`.

`link` produces a destination object from a source object, and keeps the two in sync with each other using two update procedures. Each update procedure is called when one of the objects changes, and returns the new value for the other object. When one of the update procedures has finished executing, link writes the return value to the updated object in such a way as to not trigger the other update procedure.

```
1: numtogtk :: Registry -> Number -> CB GtkItem
2: numtogtk reg (Number num) = do
3:       text <- link (lift tostring) (lift fromstring) num
4:       return $ GtkText text
```

8.3 A Translator for Group

The code below is a translator for the `Group` type. We wish to translate `Group` into the `GtkHGroup` constructor. This requires us to translate all of the members of the `Group` (which are of type `Dynamic`) into type `GtkItem`. Translators are passed a copy of the translator registry, and so are able to do this by simply applying `translate` to each of them.

```
1: grptogtk :: Registry -> Group -> CB GtkItem
2: grptogtk reg (Group l) = do
3:       mems <- mapM (translate reg "gtk") l
```

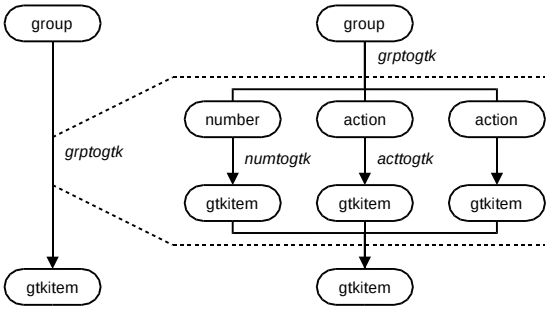

Fig. 6. Translating a `Group` - with internal translations

```
5:       let ud = map tryundy $ filtermaybe mems
6:       return $ GtkHGroup $ filtermaybe ud
```

In a more complete translator for `Group` we would want to decide what kind of group layout to use in a more intelligent way, and perhaps allow the user to adjust it, however this simplistic approach gives an easier example.

Figure 6 shows the internal and external views of this. From the outside, it appears that this translator is just translating a `Group` into a `GtkItem`. However, when one looks inside, one can see that the entire translation system is being applied recursively within the group translator.

```
link :: (a -> CB b) -> (b -> CB a) -> Var a -> CB b
lift :: (a -> b) -> a -> CB b
fromstring :: String
tostring : Int -> String
tryundy :: Dynamic -> Maybe a     -- unpack a dynamic type
filtermaybe :: [Maybe a] -> [a]   -- strip out empty items
```

9 Composing Things Together

In Feris, one is encouraged to use translators as the basic building block of modular GUIs. To demonstrate what is meant by this, we will use as an example, an application that has two instances of the `numedit` GUI, with the values of the two ints being such that one is always `10` greater than the other.

The source code for this application is given below, and a screenshot of it running is given in figure 7.

```
data TwoNum = TwoNum (Var Int)

twonum_ui :: Registry -> TwoNum -> CB VGroup
twonum_ui _ (TwoNum num) = do
   nummore <- link (lift \x -> x+10) (lift \x -> x-10) num
```

Fig. 7. The TwoNum App as a GTK GUI

```
   return $ VGroup[dy $ NumEdit num,dy $ NumEdit nummore]

twonum_app :: CB TwoNum
twonum_app = do
   num <- new 123
   return $ TwoNum num
```

One might be tempted to call `numedit_ui` directly from within `twonum_ui` rather than allowing the translator registry to arrange the call (as shown below). This should be avoided as it makes the resulting program less flexible, preventing the application being used separately from the number edit gui.

```
...  return $ VGroup[dy $ numedit_ui $ NumEdit num, ...
```

10 Some Other Targets

So far the only target we have explored is the GTK GUI target. We will now give a brief summary of some of the other targets supported by Feris. While there is not space in this paper to give detailed explanations of these targets, the information provided should be enough to give the reader a feel of how these targets work. Screenshots of the targets discussed are given in figure 8.

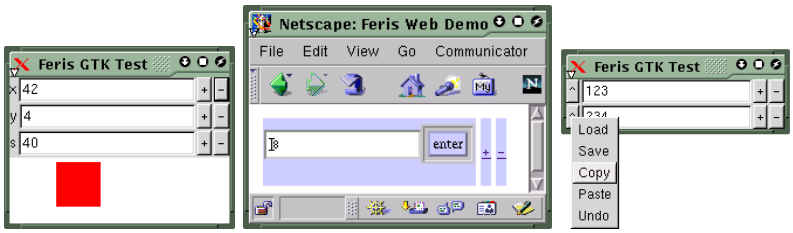

Fig. 8. Screenshots of the Graphics, Web, and File targets

10.1 Graphics

So far, we have only been working with basic controls such as edit boxes and buttons. While these are sufficient for simple examples, one often wants to be able to create a custom graphical interface.

This can be done using the graphics target. The low level type for the graphics target is given below. We use a procedure that produces a changing set of shapes, given some inputs (e.g. mouse and keyboard), and the shape of the visible area. These `GdkTarget` objects can be embedded inside GTK GUIs, using the `GtkGdk` constructor.

```
data Vlist = ...   -- List such that the contents can change
type Shapes = VList Shape
data GdkTarget = GdkTarget (Inputs -> Area -> CB Shapes)

data GtkItem = ... {- shown before -} | GtkGdk GdkTarget | ...
```

The graphics object provides a changing list of graphics primitives that are to be drawn (`VList` is similar to ListB in FranTk[13]). It is up to the GDK environment to make sure that only the minimum set of graphics primitives are redrawn, to avoid flicker during redraws, and do other similar things.

10.2 The Web

Another important target is the web. Not all interface concepts have a sensible web representation, but when there is a sensible representation, this system allows it to be used. The low level format used by the web environment is given below. Given the URL to fetch, and any form parameters, a procedure returns a procedure that provides the relevant file.

```
data FormParams = FormParams of [(String,String)]

datatype WebSite = WebSite of Url -> FormParams -> CB (CB File)
```

The layer of indirection ("`CB (CB File)`" rather than "`CB File`"), is important. The web target runs the first procedure, waits for all resultant callbacks to have executed to the user requesting that page, and then runs the second procedure to actually get the page. This is needed due to the fact that Feris callbacks are executed as atomic transactions, rather than immediately.

The UI elements used by `numedit_ui` can be easily translated into a web interface (as shown in figure 8). One will however generally get better results if one translates from something higher level. For example, one might translate `NumEdit` directly into so JavaScript[5] code that manages an editable number. One of the advantages of the Feris model is that one can do the translation at whatever level one likes.

10.3 Files

Files are treated as just another target. To allow saving of a data type, one need simply provide a translator from that data type to an array of bytes that is to be stored in the file.

Applications do not have direct access to the file system. There are no equivalents of `readFile` and `writeFile` (from the IO monad [11]). The only way in which an application can access a file is by having some of its data translated into one.

This approach gives all applications load, save, copy, paste, and undo completely for free. The environment knows how to perform these operations on files, and so can indirectly perform these operations on anything that can be translated into a file. The writer of an application does not have to think about these features in order to obtain them. For example, under a suitable environment, the `numedit_app` app is able to load, save, copy, paste and undo its number.

Inside the GTK GUI environment, it is useful to have a popup button next to any object that we can translate into a file, providing operations such as "load" and "save" (see figure 8). This requires the GTK target to be made aware of the file, using the `GtkFile` constructor of the `GtkItem` type. An enhanced translator from `Group` to `GtkItem` tries to produce a file for all of its members as well as a GUI, by applying `translate` twice.

```
data GtkItem = ... {- shown before -} | GtkFile File GtkItem | ...
```

As with other targets, one tends to get the best results if one translates from as high level a target as possible. For example, if one had a file translator for `numedit` but not `twonum`, then `twonum` would translate into a file by first being translated into a group of `numedit`s and then using the file translator for `group` to save it as two `numedit`s. This takes up twice as much space as if one were to explicitly provide a translator from `twonum` to a file.

In addition, one might want to translate to a higher level target than a raw file. For example, if one translated to a target that was able to store incremental differences between versions, then "undo" could be implemented more efficiently.

The translator approach to files also brings security advantages as applications can only access files that the user has explicitly given them. The top level environment is assumed to be trusted code, and to only access files when the user tells it to.

11 More on Callbacks

11.1 Callbacks as Transactions

A callback procedure is executed as an isolated transaction [14]. While several callback procedures may be running concurrently, they must be executed in such a way that their execution is equivalent to how it would be if only one callback ever executed at a time. This is currently implemented using a two phase locking model [14].

It should be noted that the `CB` monad does not provide the full power of the IO Monad [11]. In particular, it does not allow direct interaction with the outside world through routines like `putChar` and `getChar`. This is important, as it allows all `CB` monad operations to be rolled back if a transaction fails, and so makes a transaction based implementation practical.

Calls to the callback procedures have the following properties:

- If **react** is called several times with the same arguments, then several independent callbacks are set up.
- The callback procedure will be called exactly once for every write to the `Var` it is attached to, until the procedure returns `False`.
- After the callback returns `False`, it will not be called again.
- The argument to the callback is the value written to the `Var` in the write that the callback is reacting to.
- The callback will be called for each write in the order in which the writes took place.
- The argument to the callback is NOT guaranteed to be the current value of the `Var`. It is possible that there may have been further writes to the `Var` before the callback was serviced.

11.2 The "newsched" command

```
newsched :: CB () -> CB Sched
```

Normally, all transactions are executed in the order in which they were triggered. However sometimes this isn't what we want. Take for example the case where we want to perform a long slow execution in the background (e.g. rendering an image).

If all subsequent transactions were required to execute logically after the render had finished, then none of them would be able to commit until after the render had finished. If no callbacks commit, then no output can be made visible to the user and the system would appear to freeze. This is clearly undesirable.

One way to avoid this problem is to use `newsched`. `newsched` creates a new scheduler which executes the procedure given as its argument. Any callbacks set up by this procedure will also be executed inside this new scheduler. Note that the scheduler that a callback is executed by is determined by the call to `react` that set it up, and not by the call to `write` that set it off.

While transactions within one scheduler are constrained to execute in a fixed order, there is no fixed order for the execution of transactions in different schedulers. The system is thus free to schedule and roll back transactions so as to maximise responsiveness. In the case cited earlier, one could run the renderer in a new scheduler, allowing the the system to schedule the small operations as if they were before the big operation. The small operations would now be able to commit, at the cost of requiring the big operation to restart if a small operation wrote to something the big operation had read from.

One way to avoid such restarts of our large operation is to have a transaction that reads the input parameters, and then creates another transaction (using

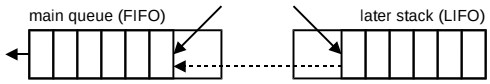

Fig. 9. The Two Scheduler Queues

`newsched`) that does the computation and writes back the result values. This allows other operations to be scheduled as being before the main operation without requiring it to be rolled back if they write to one of its inputs. The downside is that the output may be inconsistent with the current input.

11.3 The "later" command

```
later :: CB () -> CB ()
```

Sometimes one wants some code to be execute at some point later, after other callbacks have had a chance to run. One example is a program that writes to some `Var`s, and then wants to look at how other `Var` have adapted to this change.

`later` asks for a procedure to be executed when the scheduler has nothing else left to execute. This is done through the use of two scheduler queues that govern the order in which transactions are executed. Most work to be done (e.g. callbacks reacting to a `write`) is put on the main FIFO (first in first out) queue. When the main queue becomes empty, the scheduler takes something from the `later` stack. This stack is LIFO (last in first out), thus anything put onto the `later` stack is guaranteed to be executed after everything currently on the main queue, and also after any `later` procedures that are put onto the `later` stack by procedures on the main queue. This is illustrated in figure 9.

12 Related Work

The basic reactive programming model of Feris is similar to many previous systems, especially FranTk[13]. Other examples of GUI libraries for functional languages include Fudgets [3], GTK+/Haskell [18], Haggis [7] and Gadgets [9]. However, unlike these systems, Feris is based on transactions, and supports interfaces other than GUIs, through its translator system.

Outside the field of functional programming, many GUI programming systems have attempted to separate presentation from modeling. Popular examples, include COM/OLE/ActiveX [4] and Java's Swing [19]. However these are still very GUI specific and very coarse grained.

Also very relevant are XML [2] and XSL [1] which try to do similar things to Feris but with static descriptions rather than with changing interactive objects.

Acknowledgements

Enormous thanks must go to Simon Peyton-Jones, who supervised this project, and provided many useful suggestions. I would also like to thank Alan Mycroft, Søren Lassen, and Andy Gorden, who helped me with my early work in this field.

References

1. S. Adler, A. Berglund, et al. Extensible Stylesheet Language (XSL) Version 1.0 Working Draft. http://www.w3.org/TR/xsl
2. T. Bray, J. Paoli, C.M. Sperberg-McQueen, Extensible Markup Language (XML) 1.0 Specification. http://www.w3.org/TR/REC-xml
3. M. Carlsson, T. Hallgren. Fudgets - Purely Functional Processes with applications to Graphical User Interfaces. Phd Thesis, Chalmers University.
4. A. Denning. ActiveX Controls Inside Out (2nd Ed), 1997
5. B. Eich, C.R. Rand McKinney. JavaScript Language Specification (preliminary draft), 1996 http://home.netscape.com/eng/javascript/
6. C. Elliott and P. Hudak. Functional Reactive Animation. Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming 1998.
7. S. Finne, S. Peyton Jones. Composing the user interface with Haggis. Summer School on Advanced Functional Programming, 1996.
8. R. Milner, M. Tofte and R. Harper. The Definition of Standard ML. MIT Press 1997.
9. R. Noble, C. Runciman. Gadgets: Lazy Functional Components for Graphical User Interfaces. Proceedings of the 7th International Symposium on Programming Languages, Implementations, Logics and Programs, 1995.
10. S.L. Peyton Jones, J. Hughes et al. The Haskell 98 report. www.haskell.org. 1999.
11. S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. Proceedings of the 20th ACM Symposium on Principles of Programming Languages 1993.
12. S.L. Peyton Jones and John Launchbury. State in Haskell. Lisp and Symbolic Computation 1995.
13. M. Sage. FranTk - A Declarative GUI System for Haskell. http://www.haskell.org/FrantTk/userman.pdf
14. A. Thomasian. Concurrency Control: Methods, Performance, and Analysis. ACM Computing Surveys, Vol 30, March 1998.
15. T. Vullinghs, W. Schulte, and T. Schwinn. An Introduction to TkGofer. Technical Report 96-03, University of Ulm, 1996.
16. P.L. Wadler. Comprehending Monads. ACM Principles of Programming Languages, 1990.
17. The GIMP Toolkit. http://www.gtk.org
18. A GTK+ Binding for Haskell. http://www.cse.unsw.edu.au/~chak/haskell/gtk/
19. Java. http://java.sun.com