Robert Ennals

Feris: A Functional Environment for Retargetable Interactive Systems

Computer Science Part II

King's College
2000

Proforma

Name	Robert Ennals
College	King's
Title of Project	Feris: A Functional Environment for Retargetable Interactive Systems[1]
Examination	Part II Computer Science
Year	2000
Word count	10,975
Project Originator	Robert Ennals (the author)
Project Supervisor	Simon Peyton-Jones

Original Aims of the Project

To implement a system for the declarative programming of interactive applications, such that these applications may be presented as GUIs, or represented in other ways. This is to be done without allowing applications to perform actions on global system state.

Work Completed

The system was implemented. Several targets were written, including a GTK GUI target, a GDK graphics target, a web target and a file target. This allows simple programs to be written and run, and targetted at different runtime environments.

Special Difficulties

None

[1]The original title used for the project proposal was "Declarative Programming of Interactive GUI Based Applications". The name was changed during the design phase of the project, when it was decided that programs should also be able to run as web pages.

Declaration Of Originality

I Robert Ennals of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Contents

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 the following:

- Web Site
- Speech driven interface
- Command line
- Handwriting based interface (e.g. for palmtops)
- Embedded internally in another application
- Special interfaces for disabled people
- Scripting language driven interfaces
- Files[1]

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.

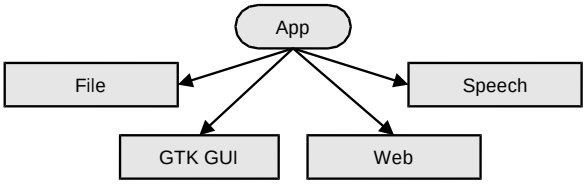

Figure 1.0.1: Multiple targets for one app module

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, and then give this bitmap to the GUI system. 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. It also makes it hard for third party programs to work with the high level data.

[1] see section 3.7.3.

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 1.0.2. 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.

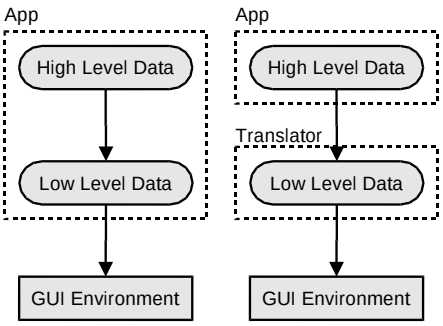

Figure 1.0.2: Moving the translation from high level data to low level data outside the application

Contributions

The key contributions of this work are the following:

- A programming model based on networks of translators, translating between interactive objects. (see section 3.5)
- The ability to retarget a single application to many types of interface, without going through a lowest common denominator. (see section 3.7)
- The ability to for a user to chose which translators and targets are to be used with an application. (see sections 3.2 and 3.3)
- 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. (see section 3.4)
- The production of a formal semantics for this monad. (see section 3.10
- A transaction system with multiple schedulers, allowing a program to easily specify constraints on transaction ordering, and transaction priorities. (see section 3.8.2)
- The ability for third parties to easily extend the system with new targets and translators.
- The production of a usable implementation of this system.

2

Preparation

2.1 Requirements Analysis

The first thing that needs to be done in the planning of a product is deciding what it is that we want the resulting system to do.

2.1.1 Targets

One of the main aims of the system is that it should be possible to write a program once, and run it in many different ways, such as as a Web page, and as a GUI driven program.

Although the system should allow programs to be run in an arbitrary number of different targets, it is only practical to implement a small number. It is important that at least two targets should be implemented, so as to demonstrate that the system does indeed allow programs to be run in multiple ways.

2.1.2 Actions on Global State

One of the core goals described in the original project proposal was that programs should not be allowed to perform actions which directly manipulate global system state. Programs should only be able to manipulate things that they have been explicitly given by the program that called them.

If programs were allowed to manipulate global state, then they could do things such as explicitly opening windows, and so couple themselves with a particular low level target (such as a GUI system), making it hard to run them in other ways.

Allowing actions on global state would also produce security issues, as a piece of code might manipulate something (e.g. a file) that we don't want it to be able to access.

2.1.3 Extensibility

It should be possible to easily add new targets and new translations to existing targets.

It should also be easy for programs written in Feris to be extended by third parties.

2.1.4 Exposing State

The system should encourage a style of programming in which high level state is exposed, and can be easily worked on by third party code.

2.1.5 Expressiveness

It is intended that the Feris model should be a practical substitute for conventional imperative approaches to programming interactive systems. In order for this to be the case, it is important that any useful[1]task that can be accomplished by a program written using an existing approach can also be accomplished by a Feris program.

2.1.6 Responsiveness

This is a special case of the previous requirement.

It is important that one should be able to execute a long operation (such as rendering an image) without causing the entire system to stall. This suggests that we need to have some support for background processing.

2.1.7 Performance

It was explicitly decided that performance should not be a requirement of the system. The goal of the project is to produce a system that demonstrates that the Feris approach is a sensible way to go about the programming of interactive systems. It is not intended that it should produce an optimal implementation.

2.2 Related Work

Before producing a design of my own, it was important to survey similar work already in existence. A summary of related work is given below.

I compare both with my project requirements, and my final design (given in section 2.3).

2.2.1 FranTk

FranTk[19] is a library for Haskell[14] that allows one to create GUIs in a declarative way. It uses a programming model loosely based on that used in Fran[7].

Similarities with Feris

- It uses a programming model based on editable variables whose changes one can react to.
- It can be used to create GUIs
- It is based around a functional programming language

Differences from Feris

- It is GUI specific.
- It does not expose high level data to the outside world.
- It allows one to perform imperative actions on global state
- Its execution model is not based on transactions

[1]We ignore exceptional cases such as viruses, where it would be good for Feris to not be able to express them.

2.2.2 COM/OLE/ActiveX

COM/OLE/ActiveX[18] is the object model used by Microsoft in their Windows[26] operating system. It provides various ways for programs to interoperate.

In particular, OLE programs may expose interfaces which other programs can use to work with their data (often used for scripting), and they may find objects in the registry which they can use to handle particular tasks.

Similarities with Feris

- It allows one to expose internal objects
- It uses a central registry to find code to handle things

Differences from Feris

- It is based around conventional imperative programming
- It is not designed to support multiple targets
- Control is in the hands of the application, rather than an environment.
- Modularity is coarser grained.

2.2.3 Intentional Programming

Intentional Programming[25] is a project being undertaken by Microsoft in conjunction with Oxford University. The aim is to allow one to easily create ones own programming language abstractions.

Similarities with Feris

- One can describe programs in arbitrary ways

Differences from Feris

- Translation is done entirely statically at compile time
- The things being translated are static programs, rather than dynamically changing objects

2.2.4 XML/XSL

XML[2] is an extensible markup language that allows one to define one's own tags with which to describe arbitrary things. XSL[1] is a language designed for translating between XML documents that use different tag sets, with the aim of producing something that a particular program (e.g. a browser) can understand.

Similarities with Feris

- One can describe things in arbitrary ways
- A form of translation system is used to convert between different documents

Differences from Feris

- The things being translated are static documents rather than changing objects

2.3 Design Decisions

After much prototyping, the following design was decided upon.

2.3.1 Translator based System Model

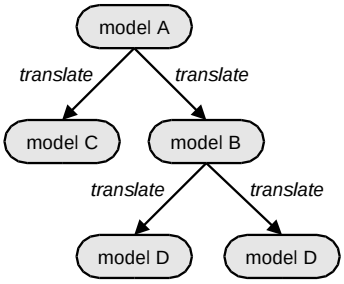

Figure 2.3.1: Translating data between different models

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 it is up to the translators to translate between them (e.g. producing pixels that represent the diagram).

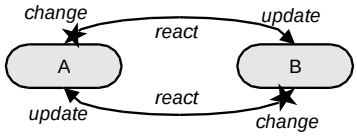

Figure 2.3.2: Keeping two models in sync with each other

Translators translate between dynamically changing objects, making sure that if one object changes, then the other is updated to take account of this change.

2.3.2 Callback based Programming Model

Translators, and other pieces of interactive code are to be produced using a programming model similar to that used by FranTk[19]. Programs can read, write and create editable variables. They can also attach callbacks to them, such that the callback procedure is executed when the associated variable is written to.

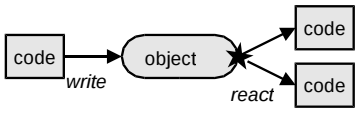

Figure 2.3.3: Callbacks React when Objects are Written to

A translator is a function that is applied once to a source object, to produce a new, destination object. When it executes, it sets up callbacks that react to changes in both objects, updating the other object so as to take account of the change. This must be done in such a way as to avoid circular updates.

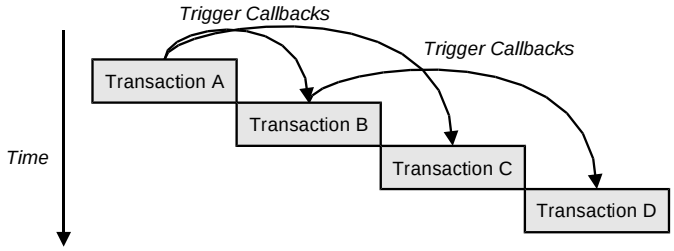

Figure 2.3.4: Callbacks executed as serialisable transactions

2.3.3 Transaction based Execution Model

As programs cannot perform global actions on state, it is possible to roll everything back. It is thus practical to implement the system as a transacting processing system.

While several callbacks may be being executed at the same time, they are executed as if they were executed strictly sequentially. This also allows the programmer to avoid the complexity of locking.

While manual locking is practical in a conventional system, where one knows what is going to be accessing ones' data, it was felt it was not appropriate for a system such as Feris, where one cannot know in advance what one might be locking against.

2.3.4 The Oval language

Oval is a language designed and implemented by the author. It is described in section 2.6. I chose to base the system on Oval due to the fact that, as I implemented it myself, I can easily extend the language and runtime as I see fit.

2.3.5 GUI and Web Targets

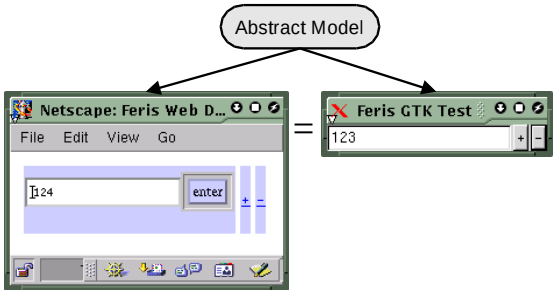

Figure 2.3.5: One App can be run on the Web, or as a GUI

Low level targets, and translators should be implemented so as to allow programs to run as both GUI apps, and Web pages.

These targets were chosen partly due to their being the most useful, and also partly because they are much easier to implement than most of the other targets discussed in the introduction. If I were to produce a speech driven target, then I would most likely end up spending most of my time on the problem of speech recognition, rather than on the core areas of the project.

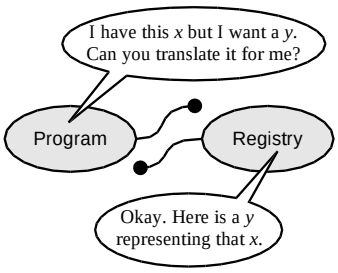

Figure 2.3.6: The Registry Provides Object Translation Services

2.3.6 Translator Registry

Translations are managed by a translator registry. If some code has an object of one type, but wants to have a representation of that object in another type, then it asks the registry to perform the relevant translation[2].

2.4 Demo Programs

In the original project proposal, it was stated that several programs should be produce to demonstrate the system, and provide milestones while the system was being developed. The demo programs are as follows:

1: Ticker

A very simple program that adds one to an integer every second.

2: Quit Button

A GUI program that does nothing, and consists entirely of a Quit button.

3: Number Edit

A program that has an integer as its data and allows one to edit this integer using increment and decrement buttons.

4: Calculator

A simple calculator in the style of the Calculator program that comes with Windows[26].

5: Mandelbrot Generator

Displays a section of the Mandelbrot set.

6: Non-blocking Mandelbrot Generator

Displays a section of the Mandelbrot set, without causing the GUI to block while the image is being generated.

[2] As illustrated in figure 2.3.6

2.5 Choice of Tools

2.5.1 GTK for the GUI target

For the implementation of the GUI target, it seems sensible to make use of an existing GUI toolkit. GTK has the advantage of being available on many different platforms, and being free.

2.5.2 Boehm Garbage Collector for Memory Management

The Oval language assumes the presence of a garbage collector. Implementing one myself would be a lot of work, so it seemed sensible to instead use the popular Boehm garbage collector. This has the advantage of being freely available and ported to many different platforms. It also claims to support POSIX threads.

2.5.3 CVS for Version Control

Some form of version control is essential in order to allow changes to be monitored and if need be, backed out. CVS has the advantage of being free, and well tested. As work is to be done on my own machine, the CVS archive will be backed up onto various other machines at regular intervals.

2.5.4 C for Implementing the Runtime

The main job the runtime has is to interface with low level libraries (such as GTK, POSIX threads and sockets). These are all designed with C programming in mind, and it seems sensible that the core runtime by written in C.

2.5.5 LaTeX for the Dissertation

LaTeX is designed for long technical documents such as dissertations. It has unrivaled support for Maths (which I may need to use), and performs many useful tasks automatically.

In order to make the dissertation look interesting, and not have the "written with LaTeX" look to it, I will create a custom style sheet.

2.6 Oval

2.6.1 The Syntax

The syntax of Oval is largely derived from Haskell[14] and ML[11]. While a full description of Oval syntax is beyond the scope of this document, the following caveats are worth noting and should be considered when reading Oval code samples.

- Function application is **right** associative. That is `f g x = f (g x)`.
- Type constructions are postfix, as with ML. Thus `bla list` is a `list` of bla, rather than a `bla` of `list`.
- The first letter of a type constructor name is capitalised, while all other words have the first letter in lower case.
- `do` notation is slightly different to that used in Haskell (see later)

2.6.2 The Module System

Oval has a module system that is quite similar to that of Java[10]. Every source file compiles to exactly one module, where the name of the module is indicated by a `package` declaration at the beginning of the file. Modules can contain functions and types[3]. Module names are hierarchical (e.g. `core.list`), and the fully qualified name of a function or type is the local name, preceded by the module name (e.g. `core.list.list` or `core.list.map`).

Symbols in the current module can be referred to by just their short name, as can any symbols that have been explicitly imported into the current module.

In the example programs given in this document, any package declarations and `import` statements are omitted for brevity.

2.6.3 The Compiler

At the time the project started, I had already written a primitive implementation of the Oval language. Implementation of this compiler had begun several months before work started on this project. The compiler is implemented in ML, and compiles Oval programs to equivalent C programs. These C programs can then be compiled to platform specific binaries using a C compiler.

Every Oval function is compiled to a C function with a name that is equal to the fully qualified name of the function, with all "." characters converted to "_" characters. For example `core.list.map` compiles to the C function `core_list_map`.

These functions use a very simple calling convention. All C functions take two arguments. The first argument is the Oval function argument (which may be a tuple or disjoint type), and the second argument contains information such as the function environment (if it was a lambda function). As part of this project, I extended the compiler to also pass around parametric type information in this argument, so as to allow the implementation of dynamic types.

2.7 Software Engineering Process

The aims of this project were highly experimental and relatively open ended. The aim was to design and construct a system to allow people to write programs in a way that is significantly different to the way in which most existing systems operate. There was thus relatively little existing work to act as a guide to how the system should be designed, and lots of opportunities to produce design flaws. As a result, it was important that a lot of time was devoted to getting the design right.

Several designs were considered, and were analysed using a combination of techniques. A formal semantics was produced for several models, giving a clearer idea of what was going on. Code for various toy programs was written in the various models in an attempt to find examples of cases where the model would fall down. For the models that survived these two passes, a simple prototype implementation was produced.

Once the main design stage had been completed, the project proceeded through a spiral model, with incremental changes being made to the design in light of experience gained from using the system.

[3] Modules cannot at present contain constants. However this would be a relatively easy feature to add.

2.8 Module Boundaries

In order to make the project more manageable, the project was split into several modules. These modules rely on a set of defined interfaces in order to interoperate with each other. For example, Web Translators depend on the web format they translate to, and the high level format they translate from. Similarly, Oval programs and the compiler both depend on the Oval language syntax.

The dependencies between the different modules are summarised in figure 2.8.1.

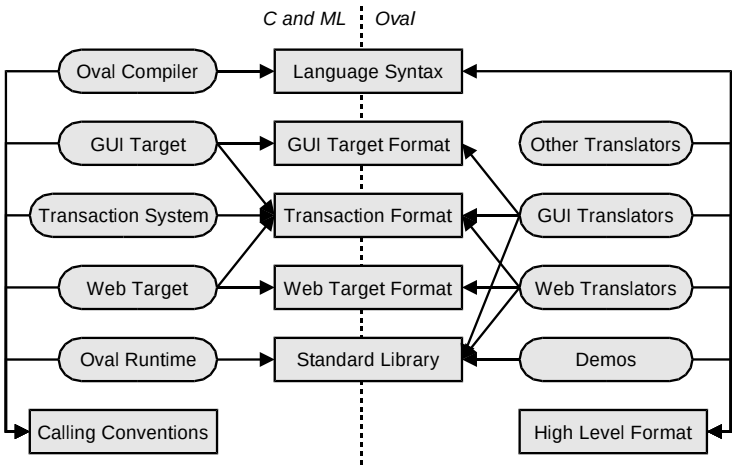

Figure 2.8.1: Module Dependencies

2.9 Testing

Testing was done largely through a system of test harness programs.

A set of programs were written that exercised various features of the system, and produced known results if the system was functioning correctly. These were accompanied by programs, written in C, that run these test programs, and checked that they produced the correct results.

Some parts of the system involved user interfaces, and these had to be tested by a human in order to make sure that they are behaving correctly.

All relevant tests were run every time a change was made to the system, in order to confirm that nothing has been broken. No changes were checked into CVS until all tests passed.

2.10 Things I had to Learn About

- LaTeX
- LaTeX style sheets
- The Boehm Garbage Collector
- Implementing Transaction Processors
- POSIX threads and synchronisation
- How web servers work

A knew a little about all of these areas before, but have to learn a lot more for the purposes of this project.

3

Implementation

3.1 The NumberEdit Demo

Possibly the best place to start a description of the implementation of Feris, is with the source code for one of the Demo programs. This provides a good overview of the general feel of the system.

Figure 3.1.1 is a screenshot of Demo 3, the Number Edit program, running in the Feris GTK[23, 13] GUI environment. It has an integer variable, and allows this integer to be increased and decreased by pressing two buttons. Code Sample 3.1.1 gives the source code for this program. The types of the functions used by this program are given in code sample 3.1.2.

Figure 3.1.1: The Number Edit App as a GTK GUI

```
 1: datatype numedit = NumEdit of int bvar;
 2:
 3: numedit_ui : registry * numedit -> group cb
 4: fun numedit_ui (_,NumEdit num) =
 5:   return Group[
 6:      dy Number num,
 7:      dy Action("+",update (num,fn x => x+1)),
 8:      dy Action("-",update (num,fn x => x-1))
 9:   ];
11
11: numedit_app : unit -> group cb
12: fun numedit_app () = do
13:   val num <- new 123;
14:   return NumEdit num;
15: end;
```

Code Sample 3.1.1: Number Edit App

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[1], and says nothing about how this editing should take place.

[1] We will see later that `int bvar` denotes an editable number within the Callback Monad.

- Line **3** is declaring the translator function[2]that is to produce a user interface for a `numedit`. The means by which this function is attached to type `numedit` is discussed in section 3.3.
- Line **5** returns as the result a group containing a number and two actions. This group is a description of a UI[3] that the environment should present to the user. Each member is a UI control that should be be displayed. Each of these items is converted to type `dynamic` using the `dy` function[4]. Note the use of right associative function application to do this.
- Line **6** defines a number item in the group. `Number` constructs a UI element that continuously displays the value of the given `int bvar`.
- Line **7** defines 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 **8** does essentially the same thing as line **6**, but creates a button to decrement the integer, rather than to increment it.
- Line **12** defines a simple top level function. Its body is a command[5]of type `group cb` that creates a numedit object for the environment to display.
- Line **13** creates a new editable number called `num`, with initial value of 123. This is the number that will be edited by the user.

```
dy : 'a -> dynamic;
update : 'a bvar * ('a -> 'a) -> unit cb;
new : 'a -> 'a bvar cb;
datatype number = Number of int bvar;
datatype action = Action of string * unit cb;
datatype group = Group of dynamic list;
```

Code Sample 3.1.2: Types relevant to NumEdit

3.2 Running Programs

Compiling a program such as NumEdit produces a loadable module file for the program. In order to actually run this program, we need to connect it up with several other modules.

A running program will generally consist of three components.

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

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.

[2]The type of this function is explained in section 3.4.

[3]Not necessarily a GUI

[4]This is needed, as the GUI elements are of different types, and yet need to be put into a list together.

[5]The `do-end` notation brackets a sequence of commands.

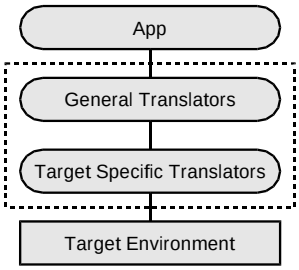

Figure 3.2.1: Modules Involved in Running an App

The target environment is a low level C program that has the job of interacting with external libraries and the operating system in order to allow interaction 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 using a set of translators. In most cases, the translators are where the majority of the work will be done.

The translator system is not just one module, but several modules that can be loaded independently. Typically one will divide up ones translators up into those specific to the target (those that deal with target specific data types), and those which are independent of any target (and just translate between different target independent data structures)[6]. However one can choose to divide up ones translators however one wants.

An important feature of this model is that the application, the translators, and the target are independent, and can be loaded in arbitrary combinations. By changing the target, one changes the means through which one interacts with the system. By changing the translators, one changes the way in which the application is translated so as to be represented on that target. There will typically be many different possible translations that can be used. For example, there are many ways that one could produce a GUI or speech interface to edit some text.

Code sample 3.2.1 gives an example of a typical top level entry point for a Feris environment. This entry point is responsible for loading the app and the translators. The way in which the application and translator registry are set up is dependent on the environment. For example, a GUI based environment might provide a GUI that allowed the user to select a program to run and to configure the set of translators[7].

```
main(){
      initruntime();
      reg = loadreg();
      app = runapp();
      gtkgui = translate(reg,"gtk",app);
      gtkdisplay(gtkgui);
}
```

Code Sample 3.2.1: Pseuedocode for a top level routine

[6]This distinction is emphasised by figure 3.2.1.

[7]Our current GUI target (for GTK) doesn't have such an app selection GUI, and requires one to edit a text file.

3.3 The Translator Registry

The Oval interface to the registry is given in code sample 3.3.1. 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 registry, and the object to translate, this function is given a string, identifying the target that we wish to translate for, such as "gtk" or "web".

The translator registry is an object that describes which translators should be used for each target that it knows about. For each target, it has a table, mapping types supported by that target to either a translator, or a "final" token.

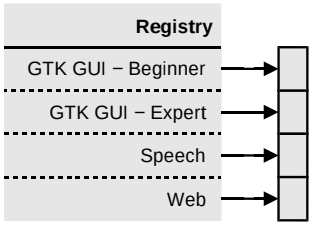

Figure 3.3.1: A translator registry can contain translator tables for several targets

Source Type	Translator
group	grptogtk
number	numtogtk
action	acttogtk
gtkitem	*FINAL*

Figure 3.3.2: Part of the translator table for the GTK GUI target

Given an object to translate for a target, `translate` will repeatedly look up the type of the object in the table for the target. If it finds a translator, then it will apply the translator to the object to get a new object, and then try to translate this for the target. If it finds a "final" token, then this is considered to be in the correct format for the target, and is returned. In the general case, one will need to apply the translation table several times in order to get something of the correct type.

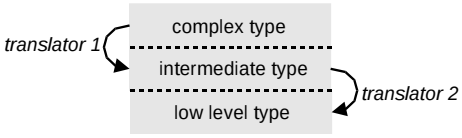

Figure 3.3.3: The registry may apply several translators in series to do a complete translation.

One may have several targets that have the same final types, but translate to it in different ways. In addition, some targets may imply one way translation instead of two way translation. This is particularly useful for files (see section 3.7.3).

The current Oval interface to the registry is given in code sample 3.3.1. As the implementation of `translate` is quite simple, it is given in its entirety in Appendix A.

For reference, the source code for `translate` is given in appendix A.

```
abstype registry;
datatype trans =
        Trans of dynamic
      | Final;

translate : registry * string * dynamic
        -> dynamic option cb;
addtranslation : registry * string * string
        * trans -> registry;
```

Code Sample 3.3.1: Interface to the Translator Registry

3.4 Editable Objects

Before we can give more examples of programs and translators, it is necessary to explain the `cb` monad. So far we have been using it in examples, but skirting over the details of what is going on. This will be a relatively quick introduction to the `cb` monad, with more detailed information being provided later.

The `cb` monad allows one to do several important things:

- Create an editable object (`bvar`)
- Read an editable object
- Write to an editable object
- Cause a callback procedure to be executed whenever an object is written to

The types and operations involved in the `cb` monad are given in table 3.4.1.

```
abstype 'a cb;
abstype 'a bvar;
abstype sched;

new : 'a -> ('a bvar) cb;
read : 'a bvar -> 'a cb;
write : 'a bvar * 'a -> unit cb;
return : 'a -> 'a cb;
seq : 'a cb * ('a -> 'b cb) -> 'b cb;

react : 'a bvar * ('a -> bool cb) -> unit cb;

later : (unit -> unit cb) -> unit cb;
newsched : (unit -> unit cb) -> sched cb;
killsched : sched -> unit cb;
```

Code Sample 3.4.1: Commands in the Callback Monad

The abstract type `'a cb` 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 -> 'b cb`, as being analogous to an imperative procedure with argument type `'a` and result type `'b`.

The abstract type `'a bvar` is an editable variable with contents of type `'a`. This is very similar to a `BVar` in FranTk [19], and to `MutVar` in the ST Monad [16]. The first five functions in figure 3.4.1 are all operations on these types, and are equivalent to functions in the ST Monad (newVar, readVar, writeVar, return, and thenST respectively). **new** creates a new bvar, and returns an identifier for that bvar. **read** obtains the current value of a bvar. **write** stores a new value in a bvar. **return** does nothing, and returns the given value.

seq allows procedures to be sequenced together to make more complex procedures. The first argument is the first procedure to execute, and the second argument is a function that takes that result of the first procedure, and returns another procedure to be executed after it.

`seq` is normally used implicitly through Haskell [14] style `do` notation. Oval `do` notation differs slightly from Haskell notation in that it requires use of a `val` keyword for variable binding, and it requires an `end` delimiter[8]. An example of Oval "do" notation, and how it translates into calls to `seq` is given in table 3.4.2.

```
fun f (a,b) = do
      val av <- read a;
      val bv <- read b;
      write (a,bv);
      write (b,av);
end;

fun f (a,b) =
      seq(read a,fn av
      => seq(read b,fn bv
      => seq(write (a,bv), fn _
      => write (b,av))));
```

Code Sample 3.4.2: An example of Oval "do" notation

react is a particularly important command. It allows one to request that a given procedure be executed when the given `bvar` is written to. The callback procedure returns a boolean value that says whether it wishes to stay attached to the `bvar`, and be executed in response to subsequent writes. There are several subtleties in the way are handled. These are addressed in section 3.8.

newsched and **later** are discussed in sections 3.8.3 and 3.8.2 respectively.

We can now define some useful functions (given in code sample 3.4.3). Some of these were used in earlier examples, and some of these are used later, when we give code for some translators. You may wish to refer to these definitions when looking at examples that use these functions.

When we say that an object of type `'a cb` is a "description" of an action, we mean just that. The functions given in code sample 3.4.1 do not perform actions when executed, but instead return a structure describing an action. The actual "execution" of these procedures is done by the transaction processing engine described in section 3.8. A formal semantics is given in section 3.10.

3.5 Translators

A translator from type `'a` to type `'b` has type `registry * 'a -> 'b cb`. The `registry` argument allows the translator to apply `translate` to any sub-objects within the object it is translating[9].

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 [23, 13] target. The GTK target requires the GUI to be described in the form given in figure 3.5.1. For reference, the types used in the result of `numedit_ui` are repeated in figure 3.5.2.

3.5.1 A Translator for Action

We will start with the translator for `action`. This takes an `action`, and produces a corresponding `gtkitem`. The code for the translator is in code sample 3.5.3.

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`'s callback procedure to be executed.

[8] Both of these changes were mode in order reduce syntactic ambiguity, and make the language easier to parse.

[9] Indeed we do this later in our translator for `group`

```
(* callback that applies a function to the
 * contents of a bvar *)

update : 'a bvar * ('a -> 'a) -> unit cb
fun update (bv,f) = do
      x <- read bv;
      write(bv,f x);
end;


(* react varient where we ignore the
 * argument, and always perform the given
 * action *)

reactalways : 'b bvar * unit cb -> unit cb
fun reactalways (bv,act) = do
      react(bv,fn _ => do
          act;
          return true;
      end);
end;


(* convert a normal function to a functon in
 * the cb monad *)

lift : ('a -> 'b) -> ('a -> 'b cb)
fun lift f = fn x => return f x;


(* map a procedure over a list *)

mapcb : ('a -> 'b cb) * 'a list -> 'b list cb
fun mapcb (f,[]) = return []
  | mapcb (f,x::xs) = do
      val y <- f x;
      val ys <- mapcb(f,xs);
      return y::ys;
end;
```

Code Sample 3.4.3: Some useful functions (for reference)

```
datatype gtkitem =
      GtkButton of string
         * unit bvar
    | GtkText of string bvar
    | GtkHGroup of gtkitem list
    ... (* further parts ommitted *);
```

Code Sample 3.5.1: The GTK GUI data type

```
datatype number = Number of int bvar;
datatype action = Action of string * unit cb;
datatype group = Group of dynamic list;
```

Code Sample 3.5.2: The types used in `numedit_ui`

The GTK Button constructor contains a string, and a `unit bvar`. The string is the name of the button, and the `unit bvar` is a variable that the button writes to whenever it is pressed. Although the bvar doesn't change value[10], 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 bvar.

```
1: acttogtk : registry * action -> gtkitem cb
2: fun acttogtk (reg,Action(name,proc)) = do
3:       val ubv <- new ();
4:       reactalways(ubv, proc);
5:       return GtkButton(name,ubv);
6: end;
```

Code Sample 3.5.3: A Translator for `action`

Let's go through the code line by line:

- Line **1** declares the type of the translator. Note that this is consistent with the general form of translator types given earlier.
- Line **2** starts the function, and obtains the name and callback procedure for the action.
- Line **3** creates a unit bvar that is to be the signal bvar for the GTK button.
- Line **4** arranges for the action callback procedure to be called every time `ubv` is written to by the button. `reactalways` is defined in figure 3.4.3.
- Line **5** constructs, and returns the final GTK button.

This illustrates the basic structure that most translators follow. They construct a new object that mirrors the source object, and link the two objects together such that an action on one object causes a corresponding action on the other object. The source and destination objects act like two different views of the same information.

3.5.2 A Translator for Number

Code sample 3.5.4 gives a slightly more complicated translator. This one deals with the `number` datatype used by `numedit_ui`. As mentioned in section 3.1, the `number` type represents an editable number.

While our set of GTK primitives doesn't contain a number editor, it does contain a basic string editor. The `GtkText` constructor describes a GTK widget that edits a `string bvar`. 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 `string bvar`. When the value of the `string bvar` changes, the GTK widget updates accordingly.

In order to use this widget to edit an `int bvar`, 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.

```
1: numtogtk : registry * number -> gtkitem cb
2: fun numtogtk (reg,Number num) = do
3:       val text <- link (num,text,lift tostring,
4:                                  lift fromstring);
5:       return GtkText text;
6: end;
```

Code Sample 3.5.4: A Translator for `number`

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 Feris has a set of standard functions[11]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[12].

3.5.3 A Translator for Group

Code sample 3.5.5 gives a translator for the `group` datatype. We translate this into the `GtkHGroup`[13]constructor. This requires us to translate all of the members of the `group` (which are of type `dynamic`) into type `gtkitem`. Fortunately, translators are passed a copy of the translator registry, and so are able to do this by simply applying `translate` to each of them. We do this using the `mapcb` function that we defined in code sample 3.4.3.

```
1: grptogtk : registry * group -> gtkitem cb
2: fun grptogtk (reg,Group l) = do
3:       val mems <- mapcb
4:           (fn x => translate(reg,"gtk",x),l);
5:       val ud = map(tryundy,filteroption mems);
6:       return GtkHGroup filteroption ud;
7: end;
```

Code Sample 3.5.5: A Translator for `group`

Figures 3.5.1 and 3.5.2 show 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.

3.6 Composing Things Together

In Feris, one is encouraged to use translators as the basic building block of modular GUIs. To demonstrate what we mean 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.

[10]It can't change value as there is only one value of type unit.

[11]Standard in the sense of being in the standard library - They are still implemented in Oval, within the `cb` monad.

[12]This is done by maintaining a flag for each link that says whether it is currently performing an update operation. All further updates through the link are blocked while this flag is set.

[13]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.

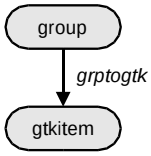

Figure 3.5.1: Translating a `group` to a `gtkitem` using `grptogtk` - The outside view

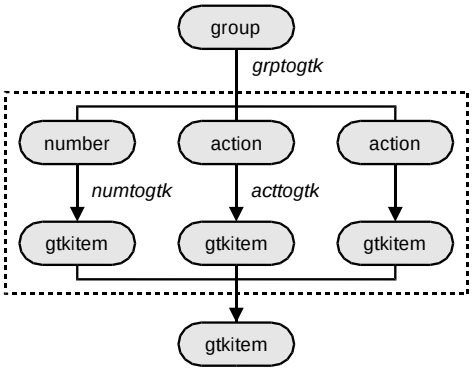

Figure 3.5.2: Translating a `group` to a `gtkitem` using `grptogtk` - The inside view

```
map : ('a -> 'b) * 'a list -> 'b list;
filteroption : 'a option list -> 'a list;
link : 'a bvar * ('a -> 'b cb) * ('b -> 'a cb)
       -> 'b bvar cb;
lift : ('a -> 'b) -> ('a -> 'b cb);
fromstring : string -> int;
tostring : int -> string;
```

Code Sample 3.5.6: Types relevant to the example translators

The source code for this application is given in code sample 3.6.1 and a screenshot of it running is given in figure 3.6.1.

Figure 3.6.1: The TwoNum App as a GTK GUI

```
datatype twonum = TwoNum of int bvar;

twonum_ui : registry * twonum -> vgroup cb
fun twonum_ui (_,TwoNum num) = do
   val nummore <- link(num,lift fn x => x+10,
                           lift fn x => x-10);
   return VGroup[dy NumEdit num,
                 dy NumEdit nummore];
end;

twonum_app : unit -> group cb
fun twonum_app () = do
   val num <- new 123;
   return TwoNum num;
end;
```

Code Sample 3.6.1: Composing Several NumEdits Together

```
datatype twonum = TwoNum of int bvar;

twonum_ui_bad : registry * twonum -> vgroup cb
fun twonum_ui_bad (_,TwoNum num) = do
   val nummore <- link (num,lift fn x => x+10,
                            lift fn x => x-10);
   return VGroup[dy numedit_ui NumEdit num,
                 dy numedit_ui NumEdit nummore];
end;
```

Code Sample 3.6.2: How Not to Compose Things

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 in code sample 3.6.2). This should be avoided as it makes the resulting program less flexible, preventing the application being used separately from the number edit gui.

3.7 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.

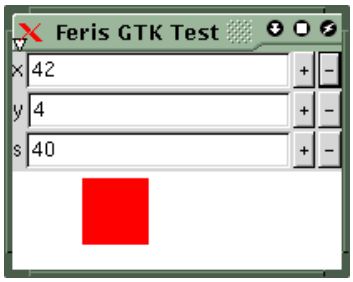

Figure 3.7.1: Using GDK inside a GTK widget

3.7.1 GDK 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 GDK graphics target. GDK is a low level graphics library designed to be used together with the GTK widget set. The low level type for the GDK target is given in code sample 3.7.1. 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[14].

```
abstype listb;
type shapes = shape blist;

datatype gdktarget =
      GdkTarget of inputs * area
         -> shapes cb;

datatype gtkitem =
      ... (* things we saw before *)
|     GtkGdk of gdktarget
      ... (* other parts omitted *);
```

Code Sample 3.7.1: The GDK target format

The `blist` type is essentially the same as the `ListB` type in FranTk [19]. It is a list of objects, such that the contents can changed.

In this case, a `blist` is used to allow the GDK object to export a changing list of GDK primitives that are to be drawn. 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.

3.7.2 The Web

Another important target is the web. The low level format used by the web environment is given in code sample 3.7.2. Given the URL to fetch, and any form parameters, a procedure returns a procedure that provides the relevant file.

```
datatype formparams =
    FParams of (string * string) list;

datatype website =
    WebSite of url * formparams -> (file cb) cb;
```

Code Sample 3.7.2: The Web Target Format

The layer of indirection ("`(file cb) cb`" rather than "`file cb`"), is important. The web target runs the first procedure, waits for all resultant transactions to have taken place[15], and then runs the second procedure. One can thus translate `abstractgui` descriptions to the web target by mapping all available actions to URLs. Given a URL, the first procedure will perform the relevant action (e.g. write to a `bvar`), and the second procedure will create a web page to describe the GUI after it has reacted to this action.

This translation is usually done using the intermediate target given in code sample 3.7.3. `webgen` procedures return a structured description of some HTML, and a table saying what operation to perform for each page the HTML references. If the `webgen` object is used inside a parent object (e.g. a `group`), then the parent adds a prefix to all of the page names referenced by the child so as to ensure that they are unique. The parent then includes the child's HTML and `optable` in its own.

```
datatype optable =
    OpTable of (string * (formparams -> unit cb))
        list

datatype webgen =
    WebGen of (html * optable) cb;
```

Code Sample 3.7.3: An Intermediate Web Target

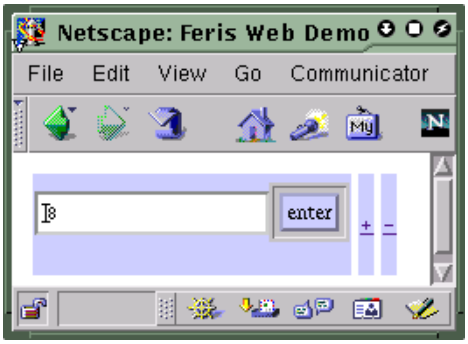

Figure 3.7.2: NumEdit as a Web Page

As one can translate `numedit` to `abstractgui`, a translation from `abstractgui` to the web target gives us a web interface for `numedit`. Alternatively, one could forget about the `abstractgui` type, and translate directly into some JavaScript [6] code that managed an editable number. One of the advantages of the Feris model is that one can do the translation at whatever level one likes.

[14] This constructor was omitted in code sample 3.5.1.
[15] See section 3.8.3 for more information on how this works.

Source code for the web target is given in Appendix C, and several example web translators are given in Appendix D.

3.7.3 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 `getChar` and `putChar` (from the IO monad [15]). 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[16] 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 popup button next to any object that we can translate into a file, providing operations such as "load" and "save". This is achieved using the `GtkFile` constructor of the `gtkitem` type (see code sample 3.7.4). 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.

```
datatype gtkitem =
    ... (* parts seen before *)
|   GtkFile of file * gtkitem
    ... (* further parts ommitted *);
```

Code Sample 3.7.4: Embedding Files into GTK

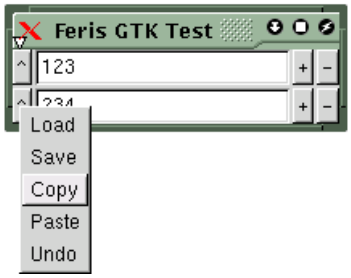

Figure 3.7.3: Things that can be translated to files get a special menu button

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[17].

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.

[17] Currently, I store different versions in full

3.8 More on the Callback System

A callback procedure is executed as an isolated transaction [20]. 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.

3.8.1 React

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 bvar 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 bvar 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 bvar. It is possible that there may have been further writes to the bvar before the callback was serviced.

For more details, refer to the semantics given is section 3.10.

These properties allow writes to bvars to be used to send messages from the writer to the reactors by writing the message value to a `bvar`.

3.8.2 The "newsched" command

Normally, all transactions are executed in the order in which they were triggered[18]. 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. This is because the renderer might write to one of the `bvar`s that the transaction had read, thus causing its execution to have been invalid. 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` (see code sample 3.8.1). `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[19].

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 it had read from.

[18] by a `write` in the case of a callback transaction.

[19] 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 triggered it.

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 `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.

The `sched` object returned by `newsched` can be passed to `killsched` to kill the scheduler.

We also simplify our implementation slightly by using only one thread per scheduler. This is a valid implementation as executions within a scheduler have to be equivalent to a sequential execution anyway. In future we may change this, so as to make better use of parallel machines.

```
newsched : (unit -> unit cb) -> sched cb;
later : (unit -> unit cb) -> unit cb;
```

Code Sample 3.8.1: Later and Newsched

3.8.3 The "later" command

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 `bvar`s, and then wants to look at how other `bvars` have adapted to this change.

One concrete example of this is the translation of the gui abstraction used by `numedit` into something usable by the web target. This translator creates a web page representing the GUI state[20]. When a button is pressed, we want to wait for any callbacks to have reacted to this, and then create a new web page based on the new state.

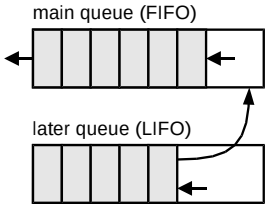

Figure 3.8.1: The Two Scheduler Queues

`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 another queue called the `later` queue. This queue is LIFO (last in first out), thus anything put onto the `later` queue is guaranteed to be executed after everything currently on the main queue, and also after any `later` procedures that are put onto the `later` queue by procedures on the main queue. This is illustrated in figure 3.8.1.

As with `newsched`, `later` is the kind of command that most programmers will never have to use, but every now and then, one comes across situations where it is extremely useful.

3.9 The Transaction System

The transaction system is part of the low level runtime, and as such is implemented in C. Some code samples are given in Appendix E.

```
\dind{trans}
struct trans{
  int pri;         /* transaction priority (lower int is higher pri) */
  dlist writes;    /* (cbwrite) writes we have performed and can roll back */
  dlist reacts;    /* (cbreq) callbacks we have set up and can cancel */
  dlist todo;      /* (outstanding) normal transactions we have scheduled */
  dlist endtodo;   /* (outstanding) ``later'' transactions we have scheduled */
  dlist locked;    /* (bvar*) bvars we have locked, and will unlock at end */
  dlist newscheds; /* (cbnewsched) new schedulers we are creating */
  pthread_cond_t* finish; /* signal when we finish */
};
```

Code Sample 3.9.1: The Transaction Structure

```
\dind{bvar}
struct bvar{
  dlist callbacks;     /* (cbreq) callbacks reacting to writes to this bvar */
  object data;         /* the current value */
  struct trans* owner; /* the transaction owning this bvar (NULL if none) */
};
```

Code Sample 3.9.2: The BVar Structure

A running transaction is described by the structure given in code sample 3.9.1.

`pri` is the priority of the transaction. In the current implementation, new schedulers created by `newsched` are considered to be inferior to their parents, and will have lower priorities.

`finish` is a condition that we signal when the transaction has either committed or aborted. Other transactions can use this to wait for the transaction to finish.

Most of the other fields contain information needed for rollback.

In order to ensure that transaction execution is equivalent to a serial execution, we use a system of strict two phase locking[20]. As a transaction (e.g. a callback) executes, it locks every `bvar` that it touches. These `bvars` are not unlocked until the transaction has either committed, or aborted.

3.9.1 Resolving Conflicts

If one transaction tries to access a `bvar` that is currently owned by another transaction, then we have a conflict. This conflict needs to be resolved in some way.

If the other transaction is superior, then we don't want to interrupt it's execution. We thus deem the current transaction to be executing logically after it, and wait for it to finish.

If the other transaction is inferior, then we don't want to allow it to interrupt the execution of the current transaction execution. We thus deem the current transaction to be executing before it, and force it to roll back.

If the other transaction is of equal priority to us, then we act as if it was superior to us, and wait for it to finish.

The low level environment (written in C) always runs at the highest scheduler priority. As a result, it never needs to worry about rolling back it's actions. This is good as coping with rollback in C code would be very awkward.

3.9.2 Rollback

One transaction forces another transaction to roll back by killing its thread, and undoing the actions it performed.

Any writes that the transaction has performed will have been recorded in the `writes` field of the transaction structure. For each write, we record the `bvar` written to, and the value that was overwritten. The thus undo by running through this list backwards. The two phase locking strategy will have ensured that none of the intermediate values were seen by other transactions.

Any callbacks that the transaction had triggered (by writing to `bvar`s), will have been recorded in the `todo` field. These are only added to the main scheduler queue when the transaction commits, and so we don't need to worry about them. Similarly for `later` transactions, which are stored in the `endtodo` field.

All `bvars` that the transaction locked are stored in the `locked` field, and are unlocked when the transaction commits or aborts.

Any callbacks that the transaction set up will be stored in the `reacts` field. These are detached from their `bvars` when the transaction is aborted. It is possible that some of these callbacks may have been triggered for execution. However the only transaction that could have triggered them is the one we are aborting, as it will have locked the `bvar` when it attached the callback to it. These will thus still be in the `todo` list, and so not get executed.

3.10 Semantics for the CB Monad

I will finish with a process based semantics for the `cb` monad.

The system state is modeled as being an unordered set of objects o, where objects are of the following forms:

o	::=	$B(v,s)_b$	A bvar b with value v, currently locked by scheduler s.
	|	$P(c,s)_p$	A running process p, executing code c on scheduler s.
	|	$C(T,r)$	A callback executing transaction T, with react id r.
	|	$N(T)$	A transaction that isn't a callback.
	|	$L(c,s,l)$	A `later` item with code c. The lth on scheduler s.
	|	$S(n,m,l)_s$	A scheduler s, currently executing its nth transaction. m and l are the indices of the next free spaces in the main, and `later` execution queues respectively.
	|	$R(b,f,s)_r$	A react item r, reacting to changes to bvar b, using callback function f, on scheduler s.
	|	T	
T	::=	$T(c,s,n)_t$	A transaction t, executing code c, on scheduler s, being the nth transaction to execute on that scheduler.

A code object can be any of the actions given in the table below. The first nine operations are the standard actions in the `cb` monad. The last three are special actions used by the semantics.

c	::=	new v	Create a new bvar, with initial value v.
	|	read b	Read the value of `bvar` b.
	|	write b v	Write value v to `bvar` b, triggering any callbacks.
	|	return v	Return value v.
	|	seq c f	Execute c followed by the result of f.
	|	react b f	React to changes in `bvar` b.
	|	later f	Execute the result of f when there is nothing else to do.
	|	newsched f	Create a new scheduler, and execute the result of f on it.
	|	killsched s	Kill an existing scheduler.
	|	trigger b v w	Trigger all callbacks on `bvar` b, except those identified in the set w, using value v.
	|	finish v	The transaction has successfully completed, with value v. All `bvars` used have been unlocked.
	|	eval e	Part-way through evaluating an Oval expression.

v represents an Oval value, f represents an Oval function, and e represents an Oval expression.

The system state makes transitions according to a set of simple rules as given in the table below. A is used as an abbreviation for an arbitrary set of other items. $\rightarrow^e$ represents an Oval expression evaluation transition.

There will usually be several transitions that can be made. Some sequences of transitions will result in the system getting stuck, due to different transactions coming into conflict with each other. It is the job of the environment to make sure that it always executes transitions such as to not get stuck, possibly making use of rollback in order to do so.

A Transaction only runs when current on its scheduler:

$$\frac{A \mid P(c,s)_p \mid S(n,m,l)_s \rightarrow A' \mid P(c',s)_p \mid S(n,m',l')_s}{A \mid T(c,s,n)_t \mid S(n,m,l)_s \rightarrow A' \mid T(c',s,n)_t \mid S(n,m',l')_s}$$

where p is new

C and N objects can behave as transactions:

$$\frac{A \mid T_t \rightarrow A' \mid T'_t}{A \mid C(T_t,r) \rightarrow A' \mid C(T'_t,r)} \qquad \frac{A \mid T_t \rightarrow A' \mid T'_t}{A \mid N(T_t) \rightarrow A' \mid N(T'_t)}$$

Seq executes its first argument:

$$\frac{A \mid P(c,s)_p \rightarrow A' \mid P(c',s)_p}{A \mid P(\text{seq } c\ f,s)_p \rightarrow A' \mid P(\text{seq } c'\ f,s)_p}$$

Executing the second seq action when the first returns:

$$\frac{A \mid P(c,s)_p \rightarrow A' \mid P(\text{return } v,s)_p}{A \mid P(\text{seq } c\ f,s)_p \rightarrow A' \mid P(\text{eval } (f\ v),s)_p}$$

Progressing with an expression evaluation:

$$\frac{e \rightarrow^e e'}{A \mid P(\text{eval } e,s)_p \rightarrow A \mid P(\text{eval } e',s)_p}$$

Executing the command resulting from an evaluation:

$A \mid P(\text{eval } c,s)_p \rightarrow A \mid P(c,s)_p$

Writing to a bvar, triggering any reacts:

$A \mid P(\text{write } b\ v,s)_p \mid B(v',s)_b \rightarrow A \mid P(\text{trigger } b\ v\ \{\},s)_p \mid B(v,s)$

Triggering a callback in reaction to a write:

$$\begin{aligned} &A \mid R(b,f,s)_r \mid P(\text{trigger } b\ v\ w,s')_p \mid S(n,m,l)_s \\ &\quad \rightarrow A \mid R(b,f,s)_r \mid P(\text{trigger } b\ v\ (w \cup \{r\}),s')_p \\ &\qquad \mid C(T(\text{eval } (f\ v),s,m)_t,r) \mid S(n,m+1,l)_s \end{aligned}$$

where t is new and $r \notin w$

When all reacts have been triggered, we can move on:

$A \mid P(\text{trigger } b\ v\ w,s)_p \rightarrow A \mid P(\text{return unit},s)_p$
if $\forall r,f,s'.(R(b,f,s')_r \in A) \Rightarrow r \in w$

Setting up a react:

$A \mid P(\text{react } b\ f,s)_p \mid B(v,s)_b \rightarrow A \mid P(\text{return unit},s)_p \mid B(v,s)_b \mid R(b,f,s)_r$
where r is new

Creating a later item:

$$\begin{aligned} &A \mid P(\text{later } f,s)_p \mid S(m,n,l)_s \\ &\quad \rightarrow A \mid P(\text{return unit},s)_p \mid S(m,n,l+1)_p \mid L(\text{eval } (f\ \text{unit}),s,l) \end{aligned}$$

Moving on to the next transaction:
$A \mid C(T(\text{finished true}, s, n)_t, r)_c \mid A \mid S(n, m, l)_s \to S(n+1, m, l)_s$

$A \mid N(T(\text{finished } v, s, n)_t) \mid S(n, m, l)_s \to A \mid S(n+1, m, l)_s$

Scheduling a later item, when we run out of transactions:
$A \mid L(c, s, l) \mid S(n, n, l+1) \to A \mid N(T(c, s, n)) \mid S(n, n+1, l)$

Ending a react when a callback returns false:
$A \mid C(T(\text{finished false}, s, n)_t, r)_c \mid S(n, m, l)_s \to A \mid S(n+1, m, l)_s$
if $\forall T, b, f.(C(T, r) \notin A) \wedge (R(b, f, s)_r \notin A)$

Removing a dead callback:
$A \mid C(T(\text{finished false}, s, n)_t, r) \mid C(T, r)_c \mid S(n, m, l)_s$
$\quad \to A \mid C(T(\text{finished false}, s, n)_t, r)_c \mid S(n, m, l)_s$

Removing a dead react:
$A \mid C(T(\text{finished false}, s, n)_t, r) \mid R(b, f, s)_r \mid S(n, m, l)_s$
$\quad \to A \mid C(T(\text{finished false}, s, n)_t, r)_c \mid S(n, m, l)_s$

Lock bvars when a transaction wants to use them:
$A \mid P(c, s)_p \mid B(v, \bot)_b \to A \mid P(c, s)_p \mid B(v, s)_b$
if $\exists v', f.\ c \in \{\text{read } b, \text{write } b\ v', \text{react } b\ f\}$

Unlock a bvar if the owner transaction has ended:
$A \mid T(\text{return } v, s, n)_t \mid B(v', s)_b \to A \mid T(\text{return } v, s, n)_t \mid B(v', \bot)_b$

Finish the transaction when everything has been unlocked:
$A \mid T(\text{return } v, s, n)_t \to A \mid T(\text{finished } v, s, n)_t$
if $\forall b, v' B(v', s)_b \notin A$

Creating bvars:
$A \mid P(\text{new } v, s)_p \to A \mid P(\text{return unit}, s)_p \mid B(v, \bot)_b$
where b is new

Reading bvars:
$A \mid P(\text{read } b, s)_p \mid B(v, s)_b \to A \mid P(\text{return } v, s)_p \mid B(v, s)_b$

Creating schedulers:
$A \mid P(\text{newsched } f, s)_p$
$\quad \to A \mid P(\text{return } s', s)_p \mid S(0, 1, 1)_{s'} \mid N(T(\text{eval } (f\ \text{unit}), s', 0)_t)$
where s' and t are new

Killing schedulers:
$A \mid P(\text{killsched } s', s)_s \mid S(n, m, l)_{s'} \to A \mid P(\text{return unit}, s)_s$

It doesn't matter if a scheduler has already been killed:
$A \mid P(\text{killsched } s', s)_s \to A \mid P(\text{return unit}, s)_s$
if $\forall n, m, l. S(n, m, l)_{s'} \notin A$

4

Evaluation

4.1 Comparison With Original Requirements

All the original requirements were met. This includes both those outlined in the original project proposal, and the extra ones added in the preparation.

Targets

The original requirement was that targets be implemented for GUI and Web applications. These were implemented, as were additional targets for GDK graphics and Files.

Actions on Global State

As originally required, the model implemented prevents any programs performing direct actions on global state.

Extensibility

The requirement was that it should be easy to add translators and targets to the system. This goal certainly seems to have been achieved. The web target and the file target were both written very quickly, without requiring alterations to existing programs. Adding a new translation is even easier, typically requiring around 3 lines of code.

It felt particularly nice when, after I implemented the file target, all my previous demos could load, save, copy, paste, and undo.

Exposing State

While it is hard to objectively evaluate a programming model, the Feris style of programming, seems to make it very easy to expose the state of one's programs.

Expressiveness

Oval is Turing complete, thus the only restriction on what can be done is the range of features provided by the available targets. While the provided set of targets does not let one do everything that an arbitrary C program could do[1], arbitrary target environments can be added, and so there seems to be no reason to believe that Feris is limited in expressiveness.

Responsiveness

`newsched` allows multiple schedulers to run in parallel, allowing one scheduler to stay responsive, while another does work in the background.

[1] E.g. there is currently no sound support.

4.2 Performance

It was explicitly decided that performance should not be a requirement for Feris. Despite this, performance seems to be quite reasonable. In most demo programs, the bottleneck is in the network or the GUI system, rather than in Feris itself. The only exception to this is the Mandelbrot demo, where the bottleneck is the speed of compiled code, but even this seems to run reasonably fast, being comparable in speed to a Mandelbrot generator running on a Java interpreter.

The main area of inefficiency in the current implementation is in the transaction processing system. This is for the following reasons:

- The `cb` monad is interpreted
- `read` and `write` have to lock their `bvar`s, even if no other transaction could possibly be using them.
- `write` has to check for callbacks

However these are all issues that can be resolved. The compiler could be modified to compile the `cb` monad into imperative code, rather than a data structure. If code was compiled dynamically, then existing visibility typing techniques could be used to determine whether a given bvar was able to escape from its scheduler, and so need to be locked[2].

The overhead of checking for callbacks could be similarly optimised away, and indeed in many cases it should be possible for a dynamic optimiser to inline the code of a callback, at the point where a write is taking place.

4.3 Test Results

All test harnesses run successfully, and indeed do so for every revision checked into CVS, as I made it a requirement that all tests passed before changes were committed. However, the output of all test harnesses was just "PASS" or "FAIL", so there is no test output that can be usefully reproduced here.

The human driven test programs produced GUIs, web pages or files. Screenshots of some of these programs are included earlier in this document.

The system appears to be very stable, and has no known bugs.

4.4 Goal Programs

At the beginning of the project, a set of goal programs were laid out. The idea was that these goal programs could be used as indicators of how the project was progressing. All of these goal programs were successfully implemented.

4.4.1 Goal 1 : Ticker

This program was successfully implemented. The source code is given in table 4.4.1.

```
count : int * 'a bvar -> int bvar cb
fun count (int,ev) = do
        num <- new 0;
        reactalways(ev, update(num,fn x => x + 1));
end;
```

Code Sample 4.4.1: Goal 1 : The Ticker

[2]Recall that the current implementation has only one thread per scheduler.

4.4.2 Goal 2 : Quit Button

The design of the GUI system evolved such as to make this essentially a "null" program. At the time this goal was proposed, it was thought that GUI system would be such that applications would send out an event when they wished to quit, however in the final design, it was decided that quitting of programs should be decided by the environment rather than by the applications themselves, and so this is simply the program that does nothing, targeted to a GUI.

One such program is given in code sample 4.4.2.

```
1: nothing : unit -> group cb
2: fun nothing () = return Group [];
```

Code Sample 4.4.2: Goal 2 : A GUI program that does nothing

4.4.3 Goals 3 and 4 : Number Edit and Calculator

Source code to Number edit was given in section 3.1.

The source code for the calculator was essentially the same, but with a few more actions, and two number inputs.

4.4.4 Goals 5 and 6 : Sync and Async Mandelbrot Generators

The source code for the asynchronous Mandelbrot generator is given in Appendix B. The source code for the synchronous Mandelbrot generator is essentially the same, except that it doesn't use "newsched" to create a new scheduler.

4.5 Code Produced

The following table summarises the code produced as part of this project.

Module	Sub-part	Lines of code	Language
Oval Compiler	**Modified**	**774**	ML
	Predating this project	7,832	ML
	Total	8,606	ML
Runtime System	`cb` Monad	1,218	C
	Misc Others	500	C
	Total	**1,718**	C
Targets	GTK GUI and Files	759	C
	GDK Graphics	489	C
	Web	408	C
	Total	**1,656**	C
Standard Library	**Misc**	**447**	Oval
Translators	**Misc**	**1,554**	Oval
Test Harnesses	**C Wrappers**	**457**	C
	Oval Test Routines	**254**	Oval

The test routines are quite small, as they largely call code from elsewhere. Translators are not catagorised as the large number of different intermediate representations means that there is no simple catagorisation. The implementation of the `cb` monad includes the transaction processor. It is not meaningful to sum line counts for C and Oval, as Oval is far more dense than C.

5

Conclusions

5.1 Comparison With Original Aims

All the original aims of the project were met. In fact, the project ended up going beyond its original aims, exploring new areas that I had not thought of at the time of the original project plan.

5.2 Evaluation of Choices Made

Looking back with the benefit of hindsight on the choices I made at the beginning of the project, there are no things I would do differently. However there were several things that caused me more trouble than I had originally expected.

5.2.1 LaTeX

I tried to push LaTeX quite hard in my dissertation, and in so doing, encountered a large number of bugs and limitations in the system. In particular, several of the packages seem to break each other. This resulted in my dissertation ending up looking less good than I had intended, and prevented me from being able to include a list of figures. Despite this, I still feel LaTeX was the right choice as there is no obvious, superior alternative.

5.2.2 Oval

The compiler ended up taking up quite a lot of my time. This was partly due to features I knew I was going to need to add (such as pattern matching), partly due to extra features I added to make it more polished (such as more friendly error messages, and a reworked syntax), and partly due to my having to rewrite several parts that turned out to be buggy. In addition, the lack of a debugger or tools for Oval made programming in it more awkward than might have been the case with another language. However I still feel that Oval was the correct choice, as working with my own language made it very easy for me to hook directly into the language runtime.

5.2.3 GTK/GDK

GTK/GDK caused problems due to its not using garbage collection, while the rest of Feris does. In particular, the Boehm GC does not scan memory allocated by GTK. This caused me to have to be rather careful about memory allocation when using GTK. There were also some problems with poorly documented functions that didn't do quite what the documentation suggested they did. In general however, GTK performed well, and was, I feel, the right tool to use.

5.2.4 The Boehm Garbage Collector

The Boehm Garbage collector claims to support POSIX threads. In fact, it supports "some" features of POSIX threads, and only then if you uncomment a well hidden comment line in a header. In particular, it corrupts itself if one tries to kill a thread that is executing the garbage collector. It took me several days to get the garbage collector and POSIX threads to work together, but it would have taken even more time to implement my own garbage collector.

5.3 Further Work

There are many further directions that this work could go in. A few are summarized below.

Translation of Translators

Translators are themselves objects within the Feris system. It might thus be sensible to apply translations to them.

This would allow there to be several different ways of describing translators, or even several different ways of implementing translations (see Lazy Translators).

The compiler itself could be considered to be a translator, from text, to a set of translators.

Lazy Translators

Translators are always applied immediately as soon as the source object changes. However, in some cases the destination object will not be read for some time, or perhaps not at all. It is thus necessary for the programmer to be careful to avoid setting up such redundant translations and so producing an inefficient program.

An alternative approach might be to have a system of lazy translators. In such a system, we would only apply translators to update a destination object when the destination object needed to be read from.

More Restricted Translator Types

The current translator type uses the `cb` monad, and so allows the translator to read from arbitrary `bvar`s at the time it is executed. If we are to have Lazy Translators, then we would want a translator execution to behave the same independently of when executed.

One way to solve this problem would be to have a more restricted translator type which didn't allow the translator to read from anything other than the source and destination objects.

Porting to Other Languages

Currently the Feris system can only be used with my Oval language. In order to reach a wider audience, it might be useful to port it to a more widely used functional programming language such as Haskell.

More Targets

I only implemented a few targets. Many more could be implemented, including such things as Speech targets, command line targets, and various different GUI targets.

Alternative Transaction Monads

The `cb` monad isn't necessarily the ideal monad to use in all circumstances. It might be interesting to experiment with a few others.

Ways of Setting up Translator Registries

Currently, the translator registry must be set up manually, by explicitly specifying what translator to use in every case.

It might be useful to have much of this done automatically, including possibly automatically adding translators contained in packages that the user has installed onto their machine.

Update Control

Currently, any procedure can write to any `bvar` that it knows about. It might be useful to have some way of saying that a procedure can read, but not write a particular set of `bvar`s.

More Efficient Implementations

The current implementation was produced with relatively little attention to efficiency. This was intentional, as the main aim of the project was to produce a proof of concept for an idea, rather than a perfect implementation of it. However if the system is to be widely used in real systems, it is necessary to produce an extremely efficient implementation of it.

Bibliography

[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] K. Claessen, T. Vullinghs, E. Meijer. Structuring Graphical Paradigms in Tk-Gofer. Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, 1997.

[5] A. Denning. ActiveX Controls Inside Out (2nd Ed), 1997

[6] B. Eich, C.R. Rand McKinney. JavaScript Language Specification (preliminary draft), 1996
http://home.netscape.com/eng/javascript/

[7] C. Elliott and P. Hudak. Functional Reactive Animation. Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming 1998.

[8] C. Elliott, Modelling 3D and Multimedia Animation with an Embedded language. Proceedings of the 1st conference on Domain Specific Languages 1997.

[9] S. Finne, S. Peyton Jones. Composing the user interface with Haggis. Summer School on Advanced Functional Programming, 1996.

[10] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Sun Microsystems, 1996.
http://java.sun.com

[11] R. Milner, M. Tofte and R. Harper. The Definition of Standard ML. MIT Press 1997.

[12] 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.

[13] H. Pennington, GTK+/Gnome Application Development. New Riders Publishing, 1999.

[14] S.L. Peyton Jones, J. Hughes et al. The Haskell 98 report. www.haskell.org. 1999.

[15] S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. Proceedings of the 20th ACM Symposium on Principles of Programming Languages 1993.

[16] S.L. Peyton Jones and John Launchbury. State in Haskell. Lisp and Symbolic Computation 1995.

[17] D. Raggett, A. Le Hors, I. Jacobs (editors). HTML 4.01 Specification http://www.w3.org/TR/html4

[18] D. Rogerson, Inside COM. Microsoft Press, 1997.

[19] M. Sage. FranTk - A Declarative GUI System for Haskell. http://www.haskell.org/FrantTk/userman.pdf

[20] A. Thomasian. Concurrency Control: Methods, Performance, and Analysis. ACM Computing Surveys, Vol 30, March 1998.

[21] T. Vullinghs, W. Schulte, and T. Schwinn. An Introduction to TkGofer. Technical Report 96-03, University of Ulm, 1996.

[22] P. Wadler. Comprehending monads. Proceedings of the ACM Sympossium of Lisp and Functional Programming 1990.

[23] The GIMP Toolkit. http://www.gtk.org

[24] A GTK+ Binding for Haskell. http://www.cse.unsw.edu.au/~chak/haskell/gtk/

[25] Intentional Programming. (not much public information as yet, but this web page is a start) http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/intentional.htm

[26] Microsoft Windows. http://www.microsoft.com

A

Source Code for “translate”

```
(* [dest = translate(reg,target,source)]
 *
 * translate <source> to <dest>, for target <target>,
 * using the translators stored in <reg> *)

translate : registry * string * dynamic
        -> dynamic option cb
fun translate (Reg xs,tgt,src) =
      case find(xs,eql,tgt) of
          SOME Tab ts => case
                  find(ts,eql,dytname src) of
              SOME Trans f => case
                      dyregapp(Reg xs,f,src) of
                  SOME act => do
                      val src2 <- act;
                      translate(Reg xs,tgt,src2)
                      end
                | NONE => return NONE
                  end
            | SOME Final => return SOME
            | NONE => return NONE
              end
       |  NONE => return NONE
          end;
```

Code Sample A.0.1: The Source Code for Translate

```
(* a translation for a type *)
datatype trans =
        Trans of dynamic
      | Final;

(* a table mapping type names to translations *)
datatype transtable =
        Tab of (string * trans) list;

(* translations for several targets *)
datatype registry =
        Reg of (string * transtable) list;
```

Code Sample A.0.2: Datatypes used by “translate”

```
(* [value = find(list,compare,key)]
 *
 * If <list> is a list of (key,value) pairs, then find the value
 * for which has a key that matches <key> when compared using <compare>
 *)

find : ('a * 'b) list * ('a * 'a -> bool)
        * 'a -> 'b option
fun find ((x,d)::xs,f,x) = if f(x,k) then SOME d
      else find(xs,f,x)
  | find _ = NONE;


(* [res = dyregapp(reg,fun,arg)]
 *
 * <fun> and <arg> are coerced out of their dynamic types.
 * then <fun> is applied to (<reg>,<arg>), giving a procedure.
 *
 * If this succeeds, then <res> = (SOME proc), where proc is the
 * resulting procedure. Otherwise <res> = NONE. *)

dyregapp : registry * dynamic * dynamic
        -> dynamic cb option
fun dyregapp (r,f,a) =
      let
          fo = tryundy f;
          ao = tryundy a;
      in case (fo,ao) of
          (SOME f,SOME a) => SOME do
              val res <- f a;
              return dy res
              end
         | _ => NONE
```

Code Sample A.0.3: Functions used by "translate"

```
abstype dynamic;

dy : 'a -> dynamic;
dyapp : dynamic * dynamic -> dynamic;
tryundy : dynamic -> 'a option;
dytname : dynamic -> string;
```

Code Sample A.0.4: Part of The Dynamic Types Interface

B

Source Code for the Mandelbrot Demo

As an example of a larger, more interesting Feris program, we give a simple asynchronous Mandelbrot generator. Features of this program include the following:

- Images are rendered incrementally, with gradually increasing resolution.
- Rendering is done in a background scheduler, so the gui remains responsive.
- Rendering restarts emmediately when the parameters are changed.

Most of the translators given here are not specific to the Mandelbrot demo, and are just generic translators that the demo happens to use.

The function `mandpix2` that calculates a mandelbrot pixel is ommitted, as it isn't very interesting, and to save space.

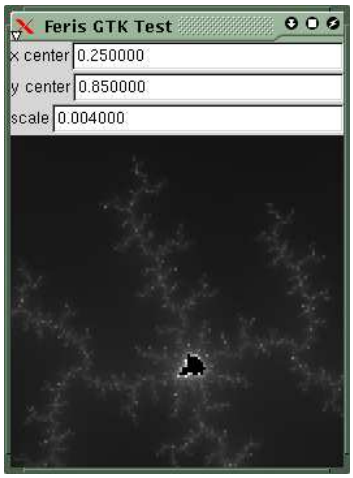

Figure B.0.1: The Asynchronous Mandelbrot Demo as a GTK GUI

```
gui : unit -> mandel cb
fun gui () = do
    val xc <- new strtofloat "0.25";
    val yc <- new strtofloat "0.85";
    val scale <- new strtof "0.004";
    return Mandel(xc,yc,scale);
end;
```

Code Sample B.0.5: The Top Level Program Description

```
(* [MultiRes (width,height,renderfunc)]
 * An image that can be rendered at different resolutions
 * by applying a function to a block size *)
datatype multires = MultiRes of int * int * (int -> rgbimage) bvar;

(* [PixelRender (width,height,pixelfunc)]
 * An image that can be rendered by applying a function to every pixel
 * coordinate *)
datatype pixelrender = PixelRender of int * int * (int * int -> color) bvar;

(* [Mandel (xcenter,ycenter,scale)]
 * A mandelbrot set, with given center, and scale. *)
datatype mandel = Mandel of float bvar * float bvar * float bvar;

(* [LabFlt(name,number)]
 * A named editable floating point number *)
datatype labflt = LabFlt of string * float bvar;

(* [EdFloat value]
 * An editable floating point number *)
datatype edfloat = EdFloat of float bvar;
```

Code Sample B.0.6: Datatypes Used by the Mandelbrot Demo

```
(* [newpixfunc = reduced (width,height,blocksize,pixelfunc)]
 *
 * Given a pixel render function, produce a new one that renders
 * at a reduced resolution *)

reduced : int * int * int * (int * int -> color) -> (int * int -> color)
fun reduced (w,h,r,f) = let
    img = makeimage(div(w,r),div(h,r),fn (x,y) => f(mult(x,r),mult(y,r)));
    f = fn (x,y) => getpoint(img,div(x,r),div(y,r));
    in f;


(* [multires = pixren_mres(reg,pixelrender)]
 *
 * Given an image described as a pixelrender, create a multires image
 * to describe it.
 *
 * We use bvmap with reduced to create a reduced res version of the
 * pixel function that is kept up to date.
 *
 * We create an image function from this pixelfunction by applying
 * makeimage. *)

pixren_mres : registry * pixelrender -> multires cb
fun pixren_mres (r,PixelRender(w,h,fbv)) = do
    val imgfunc <- bvmap(fbv,fn f =>  fn r =>
                         makeimage(w,h,reduced(w,h,r,f)));
    return  MultiRes(w,h,imgfunc);
end;
```

Code Sample B.0.7: Translator from PixelRender to MultiRes

```
labflt_gtk : registry * labflt -> group cb
fun labflt_gtk (reg,LabFlt(str,bv)) =
    return Group[dy GtkLabel str,dy EdFloat bv];

flt_gtk : registry * edfloat -> gtkitem cb
fun flt_gtk (reg,EdFloat num) = do
    val text <- link (num,lift floattostr, lift strtofloat);
    return GtkText text;
end;
```

Code Sample B.0.8: Translators from LabFlt to EdFloat, to gtkitem

```
(* If any of the paramaters changes, we react to the
 * change by updating our pixel function. *)

mand_gui : registry * mandel -> vgroup cb
fun mand_gui (_,m) = do
    val Mandel(xc,yc,scale) = m;
    val f <- mkfun m;
    val fbv <- new f;
    val act <- once upfun (m,fbv);
    reactalways(xc,act);
    reactalways(yc,act);
    reactalways(scale,act);
    return VGroup[
        dy VGroup[
           dy LabFlt ("x center", xc),
           dy LabFlt ("y center", yc),
           dy LabFlt ("scale", scale)
        ],
        dy PixelRender(256,256,fbv)
    ];
end;


(* create a pixel function for a mandelbrot set *)

mkfun : mandel -> (int * int -> color) cb
fun mkfun Mandel(xcbv,ycbv,sbv) = do
    val xcv <- read xcbv;
    val ycv <- read ycbv;
    val scalev <- read sbv;
    return mandpix2(xcv,ycv,scalev);
end;


upfun : mandel * (int * int -> color) bvar -> unit cb
fun upfun (m,fbv) = do
    val f <- mkfun m;
    write(fbv,f);
end;


(* [color = mandpix2 (xcenter,ycenter,scale) (x,y)]
 *
 * compute the mandelbrot pixel value for an image
 * coordinate, given the paramaters for this render *)

mandpix2 : float * float * float -> int * int -> color
(* implementation omitted for brevity *)
```

Code Sample B.0.9: Translator from Mandel to a GUI

```
(* [gtkitem = mmrestogtk(reg,mres)]
 *
 * Create a gtk interface to a multi-resolution image.
 * We start by showing the lowest res, and gradually get higher
 * resolutions.
 *
 * This rendering is done in a background process so as to avoid interrupting
 * other things, such as other GUI items.
 *
 * If the render function changes, then we kill our current background
 * scheduler, and create a new one, working on the new function *)

mres_gtk : registry * multires -> gtkitem cb
fun mres_gtk(_,mres) = do
    val l <- new [];
    val s <- newsched (fn () => mresrender(32,mres,Shapes l));
    val sbv <- new s;
    val MultiRes(_,_,fbv) = mres;
    reactalways(fbv,do
                val sk <- read sbv;
                killsched sk;
                val s2 <- newsched (fn () => mresrender(32,mres,Shapes l));
                write (sbv,s2);
                end);
    val white = RGB(255,255,255);
    return GtkGdk GdkTarget (white,fn _ => return Shapes l);
end;

(* [mresrender(res,multires,shapesdest)]
 *
 * Render a multires image at all resolutions from res down to
 * 1 (full res), in sequential order.
 *
 * For each resolution, we write the resultant image to shapesdest.
 *)

mresrender : int * multires * shapes -> unit cb
fun mresrender(res,mres,Shapes l) = do
    val MultiRes(w,h,fbv) = mres;
    val f <- read fbv;
    val img = f res;
    write(l,[GdkImage(img,Point(0,0))]);
    if res == 1 then return () else
        later fn _ => mresrender(div(res,2),mres,Shapes l);
end;

(* translate a normal procedure into a special procedure
 * that only executes once when scheduled to execute several times.
 *
 * We do this by maintaining a flag saying whether we have run, and
 * clearing it when the scheduler has emptied its queue.
 *)

once : unit cb -> (unit cb) cb
fun once p = do
    val flag <- new false;
    val work = do p; write (flag, false); end;
    val wrap = do
        val f <- read flag;
            if f then
                return ()
            else (do
                write (flag, true);
                later (fn _ => work);
                end);
    end;
    return wrap;
end;
```

Code Sample B.0.10: Translator for MultiRes

C

Source Code for the Web Target

Most of the code from the web target relates to web serving, and is similar to the equivalent code in other web servers, such as Apache, and AOLserver.

The only code that does anything special is in the functions `produce_page` and `main`. Both are given here.

```
/* Produce a web page in response to a client request.
 *
 * sched = the scheduler the target is running in
 * req = the url requested
 * params = any form paramaters
 * ef = a doubly indirected procedure
 *
 * Calling commit() between our calls to execute() causes the scheduler
 * to run pending transactions created by the first execute().
 */
char* produce_page(scheduler* sched, envfunc* ef, char* req, ovallist* params){
  int wrap[2] = {0, (int)params};
  int args[2] = {(int)req, (int)wrap};

  cb* proc = (cb*)APP(ef,args);
  cb* proc2 = (cb*)execute(sched,proc);
  char* page;

  commit(sched);
  page = (char*)execute(sched,proc2);
  commit(sched);

  return page;
}
```

Code Sample C.0.11: Code To Produce a Web Page

```
int main(int argc, char** argv)){
  scheduler* sched = new_scheduler(NULL); /* create a Feris scheduler */
  cb* proc = (cb*)app_main(0,0);
  int* res = (int*)execute(sched,proc);
  envfunc* ef = (envfunc*)res[1];         /* get a website object */

  commit(sched);                          /* end our transaction */
  run_server(sched,ef);                   /* start serving pages */
  return 0;
}
```

Code Sample C.0.12: The main() function for the Web Target

D

Example Translators for the Web Target

This is a small selection of the translators written for the web target.

```
acttogen : registry * action -> webgen cb
fun acttogen (r,Action(name,proc)) = let
    h = [HtmlText "<a href=",HtmlLink "x",HtmlText (">"^name^"</a>")];
    ot = OpTable [("x",fn _ => proc)];
    in return WebGen return (Html h,ot);
```

Code Sample D.0.13: Translator from action to webgen

```
numtogen : registry * number -> webgen cb
fun numtogen (r,Number num) = do
    val act = fn p => write(num,getnum p);
    val makehtml = do
           val ival <- read num;
           val html = [HtmlText "<form action=",HtmlLink "x",
                       HtmlText " method=get> <input name=num value=",
                       HtmlText tostring ival,
                       HtmlText "><input type=submit value=enter></form>"];
           val ot = OpTable [("x",act)];
           return (Html html,ot);
        end;
    return WebGen makehtml;
end;
```

Code Sample D.0.14: Translator from number to webgen

```
grouptogen : registry * bool * dynamic list -> webgen cb
fun grouptogen (r,vert,l) = do
    val mems <- mapcb(fn x => translate(r,"webgen",x),l);
    val ok = filteroption map (tryundy, filteroption mems);
    val gen = do
        val memres <- mapcb(fn WebGen x=>x,ok);
        val preres = nmap(fn (i,(h,ot))=>addprefix(tostring i,h,ot),memres,0);
        return joinwebmems preres;
    end;
    return WebGen gen;
end;
```

Code Sample D.0.15: Translator from group to webgen

E

The Transaction Processor

The transaction processing system is far too big for any substantial part of it to be presented in an Appendix. Instead a few code snippets are provided, to give a general feel for how the code works.

```
object ex_write(scheduler* s,bvar* bv,object data){
  get_bvar(&s->curtrans,bv);                    /* get a lock on a bvar */
  add_transaction_write(&s->curtrans,bv,data);  /* log rollback info and
                                                 * schedule callbacks */
  bv->data = data;                              /* write to the bvar */
  return 0;
}

object ex_seq(scheduler* s, cb* first, envfunc* fun){
  object arg = execute(s,first);                /* execute the first part */
  cb* next = (cb*)APP(fun,arg);                 /* execute the second part */
  return execute(s,next);
}
```

Code Sample E.0.16: Some code from the cb monad interpreter

```
void get_bvar(trans* t,bvar* bv){
  pthread_mutex_lock(global_mutex);          /* ensure only we are locking */
  if(bv->owner == NULL){                     /* is it unclaimed? */
    bv->owner = t;
    dlist_add(&t->locked,(int)bv);           /* add to list of things we own */
    pthread_mutex_unlock(global_mutex);
  }else if(bv->owner == t){                  /* do we already have it? */
    pthread_mutex_unlock(global_mutex);
  }else if(t->rank >= bv->owner->rank){      /* should we wait */
    pthread_cond_wait(&bv->owner->finish,global_mutex);
    pthread_mutex_unlock(global_mutex);
    get_bvar(t,bv);
  }else{                                     /* we need to roll them back */
    trans* other = bv->owner;
    transaction_kill(other);                 /* kill and roll back */
    bv->owner = t;
    dlist_add(&t->locked,(int)bv);           /* add to list of things we own */
    pthread_mutex_unlock(global_mutex);
    transaction_restart(other);              /* restart them, after us */
  }
}
```

Code Sample E.0.17: BVar locking code

Index