Porting the Clean Object I/O Library to Haskell

— Draft version (7th August 2000) —

Peter Achten[1] and Simon Peyton Jones[2]
(peter88@cs.kun.nl, simonpj@microsoft.com)

[1] Computing Science Department, University of Nijmegen, 1 Toernooiveld, 6525 ED, Nijmegen, The Netherlands
[2] Microsoft Research Cambridge, St. George House, 1 Guildhall street, Cambridge, CB2 3 NH, UK

Abstract. Pure, functional programming languages offer several solutions to construct Graphical User Interfaces (GUIs). In this paper we report on a project *in progress* in which we port the Clean Object I/O library to Haskell. The Clean Object I/O library uses an explicit environment passing scheme, based on the uniqueness type system of Clean. It supports many standard GUI features such as windows, dialogues, controls, and menus. Applications can have timing behaviour. In addition, there is support for interactive processes and message passing. The standard functional programming language Haskell uses a monadic framework for I/O. We discuss how the Object I/O library can be put in a monadic framework without loosing its essential features. We give an implementation of a small, but essential fragment of the Object I/O library to demonstrate the feasibility. We take especial consideration for the relevant design choices. One particular design choice, how to handle state, results in two versions.

1 Introduction

The pure, lazy, functional programming language *Clean* [8, 14, 18] offers a sophisticated library for programmers to construct Graphical User Interfaces (GUI) on a high level of abstraction, the *Object I/O library*. The *uniqueness type system* [20, 7] of Clean is the fundamental tool to allow safe and efficient Input/Output. This has been taken advantage of in the Object I/O library, which employs an *explicit multiple environment passing style* (a less precise but more consise term is *world as value*).

Clean uses its uniqueness type system for explicit multiple environment passing. In the functional world two other styles of solutions have been proposed to handle I/O: *stream based* and *monad based* [22, 15]. The standard functional programming language *Haskell* [12, 17] initially adopted a stream based solution up to version 1.2. From version 1.3 on monads have been firmly integrated in the language.

In this paper we report on a project (that is at this stage still in progress!) in which we investigate how to port the Clean Object I/O system to Haskell. There are several reasons to do this:

- Monads are considered to be a standard way of handling I/O in pure functional languages. It is interesting to see how well the concepts of the Object I/O system can be placed in a monadic framework.
- At this moment Haskell is lacking a standard GUI library even though several solutions have been proposed [9, 13, 19, 21]. Rather than designing 'yet another' GUI library from scratch it is worthwhile to port a GUI library to Haskell that has proven itself in practice.
- The development of the Object I/O system and Clean have mutually influenced each other beneficially. It is interesting to see if porting the system to Haskell will give a similar profit.
- The Haskell compiler that we use in this project is the GHC 4.07. It extends Haskell 98 with several features that are used in the Object I/O system such as existential types and a foreign function interface, but it also offers a treasure of simply irresistable extensions such as rank-2 polymorphism, thread creation functions, and synchronisation primitives [16]. The concurrency extensions are interesting because they allow us to implement interactive processes as true concurrent processes.

One might wonder if this project is bound to fail in advance, because Clean and Haskell use different basic techniques to bring pure functional programming and I/O into close harmony. The answer is no because even though the Object I/O system uses the world as value paradigm, it does not essentially rely on it. The key idea of the system is that GUI objects are constructed by means of algebraic data types that are parameterised by callback functions. In a world as value paradigm one can simply model these callback functions as *functions* of type $(state, *World) \rightarrow (state, *World)$. In a monadic framework these callback functions can be modelled as callback *actions* of type $state \rightarrow IOstate$.

An introduction to the basic features of the Clean Object I/O system is given in Section 2. Given its rather large size we will restrict ourselves to only a fragment of the library that contains the essential features. This is discussed in Section 3. This essential subset, the *mini Haskell Object I/O library*, is sufficiently expressive to create as a test case example a *concurrent talk* application. It is our goal in this project to implement all features required to generate this running example.

At the very outset of this project we have decided to implement two candidate versions of the mini Object I/O library. The first is a more or less one-to-one translation of the Clean Object I/O library to Haskell. The second system deviates from this system by exploiting a different way of handling state, inspired by Concurrent Haskell's `MVar`-s. This is treated in Section 4. In this section general consequences are also handled for the two systems.

The Object I/O library basically consists of two layers: at the bottom we have a layer that implements the actual interface with the underlying operating system. In this way it provides a platform independent layer upon which the other layer is constructed. This top layer is programmed completely in Clean and contains the API of the Object I/O library. The OS layer has been designed to be relatively easy to implement on most kinds of GUI toolkits. For this we

have drawn on our experience of porting earlier versions of Clean I/O libraries to platforms as Microsoft Windows, Macintosh, and X Windows. The implementation of the top layer is discussed in in Section 5. In Section 6 we discuss some of the internal implementation issues, including the OS layer.

Finally, we report on our experiences during this project with Clean and Haskell in Section 7, discuss related work in Section 8, and present conclusions and leads to future work in Section 9.

2 The Clean Object I/O system

The features of the Clean Object I/O system and its design considerations have been discussed at length elsewhere [4, 6]. Here we discuss briefly those that are relevant to this paper. For presentational reasons we use Haskell syntax augmented with Clean uniqueness annotations.

The principal concept to grasp about the Object I/O library is that it is a *state transition system*. The behaviour of every GUI object that can be defined in the library is a *callback function* that, when it needs to be evaluated, is applied to the 'current' *process state* and returns a new process state. The new process state is the next 'current' process state. The programmer only needs to define initial state values and the GUI objects that contain the behaviour functions. The Object I/O system takes care of all GUI event handling and ensures that the proper functions are applied to the proper state.

The process state consists of two parts. One part, the *program state* is defined by the programmer. It can have any type and contains the information the program needs to share between all GUI components. The other part, the so-called *I/O state*, is managed by the Object I/O system. It is a special environment and has abstract type `(IOSt ps)`, with `ps` the type of the program state. Here are the library types:

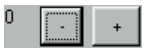
```
data *PSt  ps = PSt { ps :: !ps, io :: !IOSt ps }
data *IOSt ps -- IOSt is an abstract data type
```

For every GUI object an algebraic data type is provided that describes what that object looks like and how it behaves. Every type has a small number of mandatory arguments and a list of optional attributes. For instance, the types of *button* and *text* controls are:

```
data ButtonControl ls ps = ButtonControl String [ControlAttribute ls ps]
data TextControl   ls ps = TextControl   String [ControlAttribute ls ps]
```

In these type constructors the program state type is represented by the type argument `ps`. The `ls` parameter corresponds with the *local state* of the GUI object. As an example of local state we construct a *counter control* that displays an integer count which can be decremented and incremented with to buttons. Its definition and appearence are given below:

```
counter   = NewLS 0 (display :+: decrement :+: increment)
display   = TextControl   "0" [ControlId       displayId]
decrement = ButtonControl "-" [ControlFunction (f (-))]
increment = ButtonControl "+" [ControlFunction (f (+))]
```



The counter control is defined by `counter`. The `NewLS` data constructor encapsulates a local state of arbitrary type `new` and a collection of controls of some type `t` that work on a local state of type `new` and program state of type `ps`. `NewLS` is defined in the library as follows:

```
data NewLS t ls ps = forall new . NewLS new (t new ps)
```

The encapsulation is taken care of by *existential quantification* of `new`. The resulting expression hides the type of the local state and can be applied in any desired context. (Another version of this local state type constructor combinator is `AddLS` which we will not handle because of reasons of space.)

The individual controls are glued together by means of the `:+:` data constructor. Two expressions `e1` and `e2` can be glued together if they have types `(t1 ls ps)` and `(t2 ls ps)` respectively. That means that they must have identical local state and program state types. `:+:` is defined as follows:

```
infixr 9 :+:
data TupLS t1 t2 ls ps = (t1 ls ps) :+: (t2 ls ps)
```

Type constructor classes are used to define what elements are allowed to be glued together. To be considered a control, a type constructor must be an instance of the `Controls` type constructor class. In the Object I/O library, the following instance declarations take care that `counter` as defined above is a legal instance of this class.

```
instance Controls ButtonControl
instance Controls TextControl

instance (Controls c)              => Controls (NewLS c)
instance (Controls c1,Controls c2) => Controls (TupLS c1 c2)
```

Every GUI element has a standard, default behaviour that is determined by the rules of the underlying platform. The behaviour can be made more interesting by adding a callback function to its attribute list. For our purposes the following control attributes come into play:

```
data ControlAttribute ls ps
     = ControlFunction (IdFun *(ls,PSt ps)) | ControlId Id | ...
type IdFun x
     = x -> x
```

The behaviour of `decrement` and `increment` is very similar, so we define one *callback attribute* function `f` that is parameterised with the arithmetical operation to be performed (`(-)` and `(+)` respectively). Pressing either button

must change the local integer state and provide visual feedback to the user by changing the text of the display control. This can be done if we can *identify* the display control. For this purpose the attribute list of `display` contains an *identification attribute*. `Id` values are created by the `openId` function. They are guarenteed to be fresh with respect to each other. So the definitions of the controls and the callback function are as follows:

```
f :: (Int -> Int -> Int) -> (Int,PSt ps) -> (Int,PSt ps)
f op (count,pst)
  = (newcount,pst {io=setControlText displayId (show newcount) (io pst)})
  where
     newcount = count `op` 1
```

This explains how to define a GUI component with local state. As an example of a complete program, we can create a dialog that contains this counter control. The main function of a Clean program is called `Start`. To be interactive, it must be a function of type `*World -> *World`. The `World` environment represents the external world of a program. The library function `startIO` creates the appropriate infrastructure to do GUI I/O. Its first two arguments are the initial program state, `()`, and the function that initialises the GUI, `initialiseGUI`.

```
Start :: *World -> *World
Start world = startIO () initialiseGUI world
```

The initialisation function creates a dialog that contains the counter control that we have constructed above. The Object I/O system will now start evaluating events and determine whether one of the callback functions need to be applied. This will go on forever. The only way to terminate is by calling the library function `closeProcess`. In this case we have associated this function with the *dialog close attribute*. It will be evaluated if the user indicates in some platform dependent way that the dialog should be closed. The type of `closeProcess` is `(IdFun (PSt ps))`. However, the type of a callback function is in general `(IdFun (ls,PSt ps))`. Because this occurs very often, the convenience function `noLS` is provided that converts an `(IdFun pst)` function to an `(IdFun (ls,pst))` function.

```
initialiseGUI :: PSt ps -> PSt ps
initialiseGUI pst
    = openDialog () dialog pst1
    where
        (displayId,pst1) = openId pst
        dialog           = Dialog "Counter"
                                  counter
                                 [DialogClose (noLS closeProcess)]
```

3 A mini Object I/O library

In the previous section we have discussed the basic principles of the Object I/O library. The idea of modeling GUI programs by means of collections of state

transition objects has been exploited in the library. GUI elements that can be defined in this way are *dialogs* and *windows*. These top-level GUI elements can contain the same set of hierarchical *controls*. In the same way hierarchical and pop up *menus* have been incorporated.

The state transition concept extends in a natural way to *timers* and *receivers*. Timers are invisible elements that have a callback function that is supposed to be evaluated every given time interval. Receivers are components to which every component can send a message in a synchronous or asynchronous way. A message is considered to be an event generated by the program rather than the user or system.

Even processes can be incorporated in this model. An *interactive process* is a collection of GUI components that share the same process state (hence the name 'process') *and* share the same user interface. Interactive processes are autonomous, i.e.: one interactive process can not modify the elements of another process (such as creation or destruction of GUI elements or even termination). If it intends to do something like that it must kindly ask the other process by sending it a message (so the other process must have a receiver).

In order to show the feasibility of this project we select an essential fragment of the Object I/O library to be ported. The only top-level GUI objects to be ported are *dialogs*. The only controls to be created inside a dialog are *buttons*, *edit fields*, and *text fields*. *Interactive processes* are of course essential. We want to study how these can be implemented by means of the concurrency primitives offered by GHC. As soon as one has interactive processes, one also needs to have message passing. So we will also implement receivers and the message passing functions.

With this mini Object I/O library we can create an interesting test case: a true concurrent talk program in which two identical interactive processes are created (see Figure 1). Each interactive process offers the user a dialog with two edit fields. In the first edit field the user can type text, which is sent to the other interactive process. Received keyboard input is displayed in the second edit field.

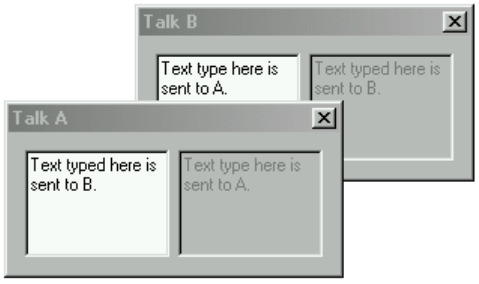


Fig. 1. Concurrent talk

4 Two Haskell Object I/O versions

In the introduction we have mentioned that we have decided to create two versions of the mini Object I/O library. The major motivation is that in our experience there is a steep learning curve before one feels at ease with the Object I/O library. This is caused mainly by the way local state is handled in combination with type constructor classes. In the Clean Object I/O library, type constructor classes are used for two purposes:

Provide an open-ended library The initial versions of the Clean GUI libraries [1, 2] used algebraic data types to enumerate the GUI elements. This has the disadvantage that one can't extend the library easily with new GUI elements without changing all the types. With type constructor classes this is no problem because you can simply add new modules when necessary. Here the type constructor class generalises the algebraic type constructor, while its instances generalise the algebraic data alternatives.

Control access to local state The composite data structures that define GUI objects contain the local state *and* the local state transition functions. The advantage of this approach is that it is statically determined what local state is accessible by what GUI component's state transition function. The disadvantage however, is that all type constructors are parameterised with two type variables, one for the current local state and one for the current global state.

To simplify matters we have decided to relax the second requirement that local state must be handled inside the GUI data structures. All state is handled by means of *synchronised mutable variables* [16] available as `MVAR`s. This design decision leads to the *dynamic* local state version of the ported Object I/O system. The 'old fashioned' version of the Object I/O system is also ported, not because we are fond of steep learning curves but to investigate its requirements on Haskell and GHC. This version is called the *static* local state version. Before we show in detail the API of the two systems in Section 5, we discuss some general consequences of this design decision.

As pointed out in the introduction, one has to decide how to model the callback functions. Instead of glueing GUI elements with state transition *functions* in Clean we will glue GUI elements with state transition *actions* in Haskell. The explicit environments that in Clean can be safely given to the programmer because the uniqueness type system protects their single threadedness are in Haskell encapsulated in a new monad for GUIs.

A callback action must be able to change the external world, so it must atleast be an `IO` monad. In addition, it must be able to perform GUI operations. As explained in Section 2, the current state of all GUI elements of an interactive process is stored in the `IOSt` record. So we need to extend the standard Haskell `IO` monad to a new `GUI` monad. For the static local state version this is done as follows: `data GUI ps a = GUI (IOSt ps -> IO (a,IOSt ps))`, such that `(GUI ps)` is an instance of the `Monad` type constructor class. For the dynamic local

state version, the GUI monad is defined as follows: `data GUI a = GUI (IOSt -> IO (a,IOSt))`, and GUI is an instance of the `Monad` type constructor class.

For both systems, the implementation, not the types, of the monad operations `>>=` and `return` are identical and straightforward:

```
(>>=) envIOA to_envIOB
    = GUI (bind envIOA to_envIOB)
    where
        bind (GUI fA) to_envIOB iost
            = fA iost >>= (\(a,iost)->case to_envIOB a of
                                        (GUI g) -> g iost)

return a = GUI (\iost -> return (a,iost))
```

In order to compare the two systems more easily, we introduce a synonym type `GUIFun` for a callback action. For the static local state it is defined as `type GUIFun ls ps = (ls,ps) -> GUI ps (ls,ps)`, and for the dynamic local state system simply as `type GUIFun a = GUI a`.

To allow easy inclusion of existing monadic code into the `GUI` monad the lifting function `liftIO` is provided (with type `:: IO a -> GUI ps a` in the state local state version, and `:: IO a -> GUI a` in the dynamic local state version):

```
liftIO m = GUI (\iost->m>>=(\a->return (a,iost)))
```

In the Clean Object I/O library two convenient lifting functions were provided when the callback function of a GUI component is not interested in its local state. This is of course relevant only for the static local state version:

```
noLS :: GUI ps a -> GUIFun ls ps
noLS action st = action >> return st

noLS1 :: (x -> GUI ps a) -> x -> GUIFun ls ps
noLS1 action x st = action x >> return st
```

5 The Haskell Object I/O API

In this section we present the API types of the static and dynamic local state versions of the Haskell Object I/O systems.

5.1 The static local state system

The set of type constructors of the *static local state system* is actually virtually identical to those discussed in Section 2. Recall that in that section, we presented the Clean Object I/O system, using Haskell syntax. We will illustrate the few differences by redoing the counter control example in Section 2.

The type constructors that define the GUI elements are identical as shown in Section 2. This implies that the definitions of `counter`, `display`, `decrement`, and `increment` do not change.

The major difference introduced by a monadic approach is, as discussed in the previous section, the use of callback *actions* instead of callback *functions*. This is reflected in the type definition of the `ControlAttribute`s:

```
data ControlAttribute ls ps
 =   ControlFunction (GUIFun ls ps) | ControlId Id | ...
type GUIFun ls ps
 =   (ls,ps) -> GUI ps (ls,ps)
```

Transforming the callback function `f` of the `decrement` and `increment` buttons to a callback action `f` results in:

```
f :: (Int -> Int -> Int) -> (Int,ps) -> GUI ps (Int,ps)
f op (count,ps)
    = setControlText displayId (show newcount) >> return (newcount,ps)
    where
        newcount = count `op` 1
```

The `Start` function is in Haskell of course the `main` function, which has the conventional `IO` monad type:

```
main :: IO ()
main = startIO () initialiseGUI
```

The GUI initialisation function is now an initialisation action. In the Clean Object I/O library its type is `(IdFun (PSt ps))`. Its monadic counterpart has type `(ps -> GUI ps ps)`. This type might look a bit strange, but it is the simple consequence of having the first type argument of `GUI` identify the global program state, and the second the result type that has to be returned. Apart from this conversion, the definition `initialiseGUI` is very similar to the one given in Section 2:

```
initialiseGUI :: ps -> GUI ps ps
initialiseGUI ps
    = do { id <- openId;
           let dialog = Dialog "Counter"
                               counter
                              [DialogClose (noLS closeProcess)]
           in  openDialog () dialog >> return ps;
      }
```

The concurrent talk example: *to do*.

5.2 The dynamic local state system

The main feature of the dynamic local state version of the Haskell Object I/O system is the way state is handled. Instead of relying on the type constructors to pass around the proper (local and global) state, the mutable variable concept of Concurrent Haskell is used. The first noticable consequence of this decision

is that the type constructors become less cluttered because there are no more state parameters. For instance, this is what becomes of the GUI element type constructors:

```
data ButtonControl    = ButtonControl String [ControlAttribute]
data TextControl      = TextControl   String [ControlAttribute]

data ControlAttribute = ControlFunction (GUIFun ()) | ControlId Id | ...
```

Other simplifications that follow are the elimination of the local state type constructor combinators `NewLS` and `AddLS`. The type of the `:+:` type constructor combinator also profits, and becomes simply `data TupLS t1 t2 = t1 :+: t2`.

A nice property is that even though the type constructors are clearer, one can still write exactly the same GUI objects in a program. So the definition of `counter` is the same in Clean as well as in the Haskell static and dynamic local state versions. What is different is of course the way callback actions handle logical state. This is what is handled in the remainder of this section.

Mutable variables, as introduced in Concurrent Haskell, have the following signature (these are only the basic operations):

```
data MVAR a -- MVar is an abstract data type
newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

A value of type `(MVar a)` is a reference to a mutable location that is either *empty* or *filled* with a value of type `a`. The functions `newEmptyMVar` and `newMVar` create an *empty* and a *filled* `MVar` respectively. The content of a filled `MVar` can be retrieved using the function `takeMVar`, which returns the content and empties the `MVar`. An empty `MVar` can be filled with a value of the proper type with the function `putMVar`.

`MVars` have been designed to work in a concurrent setting. This becomes apparant when applying `takeMVar` and `putMVar` to arguments that are empty or full respectively. In these cases, the functions *block* and can proceed only if *another concurrent process* fills or empties the `MVar` respectively.

Finally, when there are no more references to a created `MVar`, the mutable location is automatically garbage collected.

The consequences for the counter control example are the following. The callback action of the `decrement` and `increment` buttons, `f` becomes (assuming that identifier `c` has type `(MVar Int)`):

```
f :: (Int -> Int -> Int) -> GUIFun ()
f op = do { count <- takeMVar c;
            let newcount = count `op` 1
            in  setControlText displayId (show newcount) >>
                putMVar c newcount;
      }
```

The initialisation action `initialiseGUI` in addition to creating the dialog, must also create the initial local state `MVar`. We choose to do this the following way:

```
initialiseGUI :: GUI ()
initialiseGUI
    = do { id <- openId;
           let counter = do { c <- newMVar 0;
                              let display   = ...
                                  decrement = ...
                                  increment = ...
                              in  return display:+:decrement:+:increment;
                            }
               dialog  = Dialog "Counter"
                                counter
                              [DialogClose closeProcess]
           in  openDialog () dialog;
      }
```

The following observations can be made from this example:

- It illustrates the way one can use *lexical scoping* to achieve a similar hiding effect as existential encapsulation does in the static local state version. The `MVar c` is known only in the context where it is supposed to. This way of handling references to objects has a precedence in the Object I/O library, namely in the way `Id` values are created and passed around. This is visible in the very first action of the program where the function `openId` is used to create the `Id` value for the `display` control.
- In the static local state version the relationship between (local and global) state and the state transition actions is determined by the type constructor combinators. When a transition must be computed, the Object I/O system knows what state to apply to a transition action, and where to update the administration with the new state. This does not apply in the dynamic local state version. It is the responsibility of the programmer to manage the state references properly.

Handling state in this way increases the expressive power of the system. In the dynamic local state system one can construct programs that can't be created in the static local state system (without adding more complicated type constructor combinators if at all). As an example, consider a program that has three dialogs, D_1, D_2, and D_3 such that D_1 and D_2 have a state A in scope, and D_1 and D_3 have a state B in scope:

```
main = do { a <- openD2;
            b <- openD3;
            openDialog (d1def a b)
      }
     where
```

```
openD2 = do { x <- newMVar v1;
              openDialog (d2def x);
              return x
            }
```

The concurrent talk example: *to do*.

6 Implementation issues

In this section we will discuss the most interesting aspects of the implementation of the two versions to Haskell. *Note:* at the time of writing this project was not yet finished, and we have just started the implementation phase. As a consequence this section does not report work that has been done but instead how we have planned to tackle some of the problems. The current state of the implementation is that we have the framework done for both versions, i.e.: all the API type modules and most of the underlying modules have been ported to Haskell. For the two versions of the Haskell Object I/O systems, most implementation issues are identical. In fact, by using the GHC C pre-processor, we are working on one source library for the two versions. In most cases, the only differences are related to the types.

The implementation of the static local state system differs considerably from the dynamic local state version in the way event handling triggers the state transition computations. This is discussed in Section 6.1. The Object I/O library has always been designed with concurrency in mind: interactive processes should, in principle, be able to run concurrently, and even be distributed on a network. This is discussed in Section 6.2. The concurrent talk example relies on message passing. In Section 6.3 we present how this can be implemented. Finally, in Section 6.4 we show how the OS dependent layer will be implemented.

6.1 Static local state implementation issues

As explained in Section 2, the Object I/O system is a state transition system. The GUI elements contains callback actions, which internally are state transition functions. For the dynamic local state system, a callback action is a simple `GUI` monad, being an `IO` monad extended with the `IOSt`. Nothing special is happening here. For the static local state system calculating local state transitions is somewhat more complicated.

The Clean Object I/O implementation uses an elegant lazy solution to compute and store local state in the `IOSt` [5]. The crux of the idea is as follows: the programmer writes local state transition functions, which have type `(ls,PSt ps) -> (ls,PSt ps)`. Now, recall that the `PSt` is a record that contains the `IOSt` which is the data structure managed by the Object I/O runtime system that contains *all* information of the GUI. *All* includes also the local state `ls`. So, the task of the Object I/O runtime system is to provide the local state transition function with the correct local state and process state and to restore the new

local state in the process state. Let `f` be the callback function of the given type above. What happens is its actual local state argument is a reference to the original local state value, while the process state argument is the process state that is updated with the *new* local state value computed by `f`.

Because Clean is a graph rewriting language (although one might not see this immediately due to syntactic sugar) and it uses an explicit environment passing paradigm this idea of lazy forward references is a natural solution. This is less so in the 'imperative' nature of the monadic way of handling I/O in Haskell. Recall that a callback action has type `(ls,ps) -> GUI ps (ls,ps)`. Here, the `IOSt` record is hidden inside the `GUI` monad. So the task for the Haskell Object I/O runtime system is to apply this monadic action in such a way that the returned `(ls,ps)` state pair can be updated in the internal `IOSt` record *before* the action is applied. This sounds like a contradiction. Fortunately, a solution exists, namely by means of the monadic extension `fixIO` which has type `fixIO :: (a -> IO a) -> IO a`. Its argument function is applied to the result of its action, which is what is required. *Actual implementation to be done.*

6.2 Interactive processes

The concurrent talk test case spawns two interactive processes. Note that there is no semantic problem with having interactive processes in the Object I/O system. This has been explained in more detail in [3]. Semantically, interactive processes can be computed sequentially by interleaving their callback functions. A callback function modifies its environment, which is then passed to the next, and so on. To allow a concurrent semantics, one only needs to introduce a context-switch at those I/O operations that can in principle affect other interactive processes (writing to the file system for instance). In this way the semantics allows one process to interfere with another *during* the evaluation of a callback function. This is of course necessary to obtain a natural semantics sharing external resources (such as the file system), but also for message passing.

Apart from sharing the external world, interactive processes are autonomous (see the discussion in Section 3). Given the considerations above it follows that it is legal to implement interactive processes as truely *concurrent* entities. For this purpose we will use two primitives offered by GHC:

```
forkIO     :: IO () -> IO ThreadId
killThread :: ThreadId -> IO ()
```

Actual implementation to be done.

6.3 Message passing

Every GUI object can send messages to special GUI objects, called receivers. This can be done asynchronously and synchronously. When sending messages synchronously, the receiver can also reply with a return message. To make sure sender and receiver agree on the type of the messages, special receiver identification values are provided in the library that are parameterised with the message

type. A receiver that handles messages of type `m` is identified by a value of type `(RId m)`. A bi-directional receiver that handles messages of type `m` and replies with messages of type `r` is identified by a value of type `(R2Id m r)`.

We intend to implement message passing for both versions of the Haskell Object I/O system by reusing an abstraction built on top of `MVar`s, called *channels* [16]. *Actual implementation to be done.*

6.4 The OS layer

In Section 1 we have stated that we want to reuse the existing C code that has been written for the Clean version of the Object I/O OS dependent layer. For Haskell 98 the *Foreign Function Interface* [10] has been proposed to be able to write down Haskell code that calls upon foreign functionality. There are several problems that need to be worked out:

- The Clean C call interface allows Clean functions that call C code to return tuples of basic type. The corresponding C call gets these parameters passed as pointers to these basic values. In some way, code must be added in the Haskell implementation to call these C procedures properly.
- All Clean OS functions are basically `*OSToolbox` transition functions. This environment conceptually models the GUI toolkit. It is conveniently implemented as an integer. In a monadic setting this value is in principle superfluous, as sequentialisation has already been realised within the monad.
- Clean *String*s are arrays of unboxed `Char`s. In Haskell, they are implemented as `[Char]`. We have to decide what the best place is to convert these data types.

Despite these problems, we expect that it is relatively easy to connect Haskell with the C code that has been written for the Clean Object I/O platform.

7 Experience

In this section we will report our experience with the differences between Clean and Haskell and their respective compilers. To do.

8 Related work

To do. To our knowledge this is the first high level GUI library for functional languages that provides ports to more than one functional language.

9 Conclusions and future work

No conclusions yet as we're in the middle of this project. Of course we do have a vision on the near future. If this project turns out to be successfull, we can

decide how to continue. Some of the design decisions will have their impact on the Clean Object I/O library. We hope to find some resources to create a port of the full Object I/O library to Haskell. Ideally, we should end up with a library that allows functional programmers to share code more easily between Haskell and Clean. At least one interesting, but ambitious, application is to see how well the code of the Clean IDE will behave on the Haskell port.

Acknowledgements

Peter would like to thank Simon Marlow, Julian Seward, and Rueben Thomas for their advice and help on Haskell and the GHC compiler, Andrei Serjantov for showing him around in Cambridge, and the people at Microsoft Research Cambridge for providing a stimulating working environment.

References

[1] Achten, P.M., van Groningen, J.H.G., and Plasmeijer, M.J. High Level Specification of I/O in Functional Languages. In Launchbury, J., Sansom, P. eds., *Proceedings Glasgow Workshop on Functional Programming*, Ayr, Scotland, 6-8 July 1992. Workshops in Computing, Springer-Verlag, Berlin, 1993, pp. 1-17.

[2] Achten, P.M. and Plasmeijer, M.J. The ins and outs of Clean I/O. In *Journal of Functional Programming* **5**(1) - January 1995, Cambridge University Press, pp. 81-110.

[3] Achten, P.M. and Plasmeijer, M.J. Concurrent Interactive Processes in a Pure Functional Language. In Vliet, J.C. van, ed., *Proceedings Computing Science in the Netherlands, CSN'95*, Jaarbeurs Utrecht, The Netherlands, November 27-28, Stichting Mathematisch Centrum, Amsterdam, 1995, pp. 10-21.

[4] Achten, P.M. and Plasmeijer, M.J. Interactive Functional Objects in Clean. In Clack, C., Hammond, K. and Davie, T. eds., *Proceedings 9th International Workshop Implementation of Functional Languages, IFL'97*, St. Andrews, Scotland, UK, September 1997, selected papers, LNCS **1467**, Springer, pp. 304-321.

[5] Achten, P.M. and Plasmeijer, M.J. The implementation of interactive local state transition systems in Clean. In Koopman, P., Clack, C. eds., *Proceedings 11th International Workshop Implementation of Functional Languages, IFL'99*, Lochem, The Netherlands, September 1999, selected papers, LNCS **1868**, Springer, pp. 115-130.

[6] Achten, P.M. and Wierich, M. A Tutorial to the Clean Object I/O Library - version 1.2. *Technical Report CSI-R0003*, February 2, 2000, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, The Netherlands.

[7] Barendsen, E. and Smetsers, J.E.W. Uniqueness Type Inference. In [11], pp. 189-206.

[8] Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, and Plasmeijer, M.J. Clean: A Language for Functional Graph Rewriting. In Kahn. G. ed. *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.

[9] Carlsson, M. and Hallgren, Th. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, 9-11 June 1993, ACM Press, pp. 321-330.

[10] Finne, S. A Haskell foreign function interface. Available as GHC extension, see: `http://www.haskell.org/ghc/docs/latest/set/sec-foreign.html`. (*Better reference needed, sorry.*)

[11] Hermenegildo, M. and Swierstra, S.D. eds. *Proceedings of Seventh International Symposium on Programming Languages: Implementations, Logics and Programs*, Utrecht, The Netherlands, 20-22 September 1995, LNCS **982**, Springer-Verlag.

[12] Hudak, P., Peyton Jones, S., Wadler, Ph., Boutel, B., Fairbairn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, Th., Kieburtz, D., Nikhil, R., Partain, W., Peterson, J. Report on the programming language Haskell, In *ACM SigPlan Notices*, **27** (5): 1-164.

[13] Noble, R. and Runciman, C. Gadgets: Lazy Functional Components for Graphical User Interfaces. In [11], pp. 321-340.

[14] Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. Concurrent Clean. In Aarts, E.H.L., Leeuwen, J. van, Rem, M., eds., *Proceedings of Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202-219.

[15] Peyton Jones, S.L. and Wadler, Ph. Imperative Functional Programming. In *Proceedings of the Twentieth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 10-13, 1993, pp. 71-84.

[16] Peyton Jones, S., Gordon, A., Finne, S. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, January 1996, St.Petersburg Beach, Florida, ACM, pp. 295-308.

[17] Peyton Jones, S. and Hughes, J. eds. *Report on the Programming Language Haskell 98 – A Non-strict, Purely Functional Language*, 1 February 1999.

[18] Plasmeijer, M.J. and van Eekelen, M.C.J.D. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Publishing Company 1993.

[19] Scholz, E. PIDGETS - Unifying Pictures and Widgets in a Constraint-Based Framework for Concurrent GUI Programming. In Kuchen, H., Swierstra, S.D. eds. *Proceedings of eighth International Symposium on Programming Languages: Implementations, Logics, and Programs*, Aachen, Germany, September 1996, LNCS **1140**, Springer, pp. 363-377.

[20] Smetsers, J.E.W., Barendsen, E., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In Schneider, H.J., Ehrig, H. eds. *Proceedings Workshop Graph Transformations in Computer Science*, Dagstuhl Castle, Germany, January 4-8, 1993, LNCS **776**, Springer-Verlag, Berlin, pp. 358-379.

[21] Vullinghs, T., Schulte, W., and Schwinn, Th. The Design of a Functional GUI Library Using Constructor Classes. In Bjørner, D., Broy, M., Pottosin, I.V. eds. *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference Akademgorodok*, Novosibirsk, Russia, June 25-28, 1996, Proceedings, LNCS **1181**, Springer, pp. 398-408.

[22] Wadler, Ph. Comprehending monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, Nice, ACM Press, pp. 61-78.