# Porting the Clean Object I/O Library to Haskell

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 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 an 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* [9, 17, 21] 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* [24, 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 concise term is *"world as value"*). From the outset on [1, 2] one of the key features of the Clean I/O project has been the explicit handling of *state*, and the specification of graphical user interfaces at a high level of abstraction. The approach has proven to be successful and flexible, allowing the model to be extended with interactive processes (on an interleaving and concurrent basis [3]), message passing (synchronous and asynchronous), and local state resulting in an object oriented style [4, 5]. The library provides a rather complete set of GUI objects for real-world applications and produces efficient code. This has been demonstrated by writing a complete integrated development environment, the *CleanIDE*.

In Clean the uniqueness type system is used to support I/O in an explicit multiple environment passing style. Two other styles of solutions have been proposed to handle I/O in a purely-functional setting: *stream based* and *monad based* [27, 18]. The standard functional programming language *Haskell* [15, 20] initially adopted a stream based solution up to version 1.2. From version 1.3 on monads were firmly integrated in the language. Many interesting experimental frameworks have been proposed to handle GUI programming in both styles ([10, 16, 23, 25] to name a few). For a broad overview see Section 7.

In this paper we report on a project in which we ported a core subset of this I/O system to Haskell. There are several motives to embark on such a project.

- Monads are considered to be a standard way of handling I/O in pure functional languages. In this project we demonstrate that it is possible to transfer the concepts of the Object I/O system to a monadic framework.
- Designing a solution to functional GUI programming is one thing, but it is a truly large effort to maintain, extend and improve such a system. The Clean Object I/O library has proven itself in practice. It is efficient and in a fairly stable state. Porting this library to Haskell is a relatively small effort.
- When comparing programming languages and the applications written in them, it is crucial to share identical libraries. Especially for the important application domain of interactive applications, the lack of these libraries makes it hard to do serious comparative studies.
- The development of the Object I/O system and the Clean language have mutually influenced each other beneficially. One can expect similar effects between library and language when porting the system to Haskell.
- The Haskell compiler that we use in this project is the Glasgow Haskell Compiler 4.08.1. It extends Haskell 98 with several features that are required by the Object I/O system (*existential types* and a *foreign function interface*). In addition to these features it supports a variety of useful extensions such as *rank-2 polymorphism*, *thread creation*, and *communication/synchronisation* primitives. In this project we show how we have used these extensions to simplify the implementation of interactive processes and message passing.

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 described by algebraic data types. The *behaviour* of a GUI object is defined by a set of *callbacks*. A callback is essentially a piece of code that must be executed in well-defined circumstances (usually called *events*). In a world as value paradigm one can simply model these callbacks as *functions* of type `(state,*World) -> (state,*World)`. In a monadic framework these callbacks can be modeled as *monadic actions* of type `state -> IO state`, or even just `IO ()`.

A closely related question is whether it is possible to handle *local state* in a monadic framework in a way that reflects the philosophy of the Object I/O library. We show that it is possible to provide a translation to Haskell that (except for the obvious difference in callbacks) is exactly identical to the Clean version. However, we also explored an alternative design, in which state is held in mutable variables, an approach that turns out to give a considerably simpler type structure. Because the local state version of the Object I/O library has been discussed at length elsewhere [4, 6], we will discuss the alternative mutable variable based design in full detail in this paper, and compare it with the local state version in Section 3.

The Clean Object I/O library is **big**. Version 1.2.1 consists of 145 modules that provide an *application programmer's interface* (API) of 43 modules giving access to roughly 500 functions and 125 data types. For a feasibility study this is obviously a bit to much to port, so we have restricted ourselves to a fragment of the API that contains the essential features. This subset, the *mini Haskell Object I/O library*, is sufficiently expressive to create as a test case target a *concurrent talk* application (see Figure 1(a)). In the mini Haskell Object I/O library you can open and close arbitrarily many *interactive processes* (two in the test case). Each interactive process can open and close arbitrarily many *dialogues* (one in each interactive process). Each dialogue can contain arbitrarily many *text-*, *edit-*, and *button controls* (in the test case the dialogues contain two edit controls, one for input, one for output). In addition we have ported *asynchronous message passing* (text typed in the upper edit control is sent to the receiver of the other interactive process which displays the text in the lower edit control).

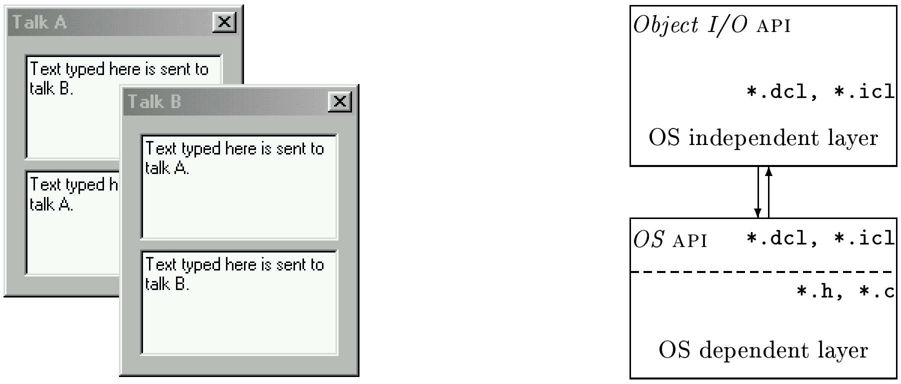



**Fig. 1. (a)** Concurrent talk **(b)** Layered architecture

Another means of reducing the porting effort to Haskell is by making use of the layered architecture of the Clean Object I/O library (see Figure 1(b)). 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. This is the OS *dependendent layer*. It defines an interface that is used by the top layer, which is therefore OS *independent*. The OS independent layer is written entirely in Clean. The OS dependent layer has been designed in such a way that it is 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 remainder of this paper is structured as follows. We start with a detailed discussion of the mutable variable based version of the mini Haskell Object I/O library API in Section 2. We then compare this new approach with a one-to-one translation of the Clean Object I/O library to Haskell in Section 3. The implementation of the OS independent layer (Section 4) and the OS dependent layer (Section 5) are basically the same for both versions. Porting the Clean Object I/O library to Haskell is a good opportunity to compare the two languages, libraries, and tools. This is done in Section 6. We present related work in Section 7 and conclude in Section 8.

## 2 The mini Haskell Object I/O API

As our first step, we present the design of the mini Haskell Object I/O system, as seen by the programmer. The version presented in this section handles state by means of mutable variables. The design rationale is basically the same as the local state version of the Object I/O library, so we will not discuss these. Instead we content ourselves with a brief overview based on examples.

As has been argued briefly in the introduction, the only true language independent difference between the two libraries is the way callbacks are represented. We give the monadic approach in Section 2.1. Then we illustrate the way local state is handled in Section 2.2. The remaining essential GUI components that are required for the concurrent talk test case are handled in Section 2.3.

### 2.1 A monad for state transitions

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* 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.

In the Clean Object I/O library, the process state is handed to the programmer explicitly as environment value of abstract type `IOSt` (called the *I/O state*). This environment is managed entirely by the Object I/O system. Every callback is forced by the uniqueness type system of Clean to return a unique I/O state. As the I/O state is an abstract value, and there are no denotations available to the programmer we can ensure that all GUI operations can be performed safely.

In Haskell, instead of passing the I/O state around explicitly, we encapsulate it in a monad, in the standard way:

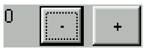
```
data GUI a = GUI (IOSt -> IO (a,IOSt))}

instance Monad GUI where
    (>>=)  = bindGUI
    return = returnGUI

bindGUI :: GUI a -> (a -> GUI b) -> GUI b
bindGUI (GUI fA) to_ioB ioSt
  = GUI (\ioSt -> do { (a,ioSt1) <- fA ioSt ;
                       case to_ioB a of
                         GUI fB -> fB ioSt })

returnGUI :: a -> GUI a
returnGUI a = GUI (\ioSt -> return (a,ioSt))
```

Defining the `GUI` monad to be an *enhanced* `IO` monad allows us to combine existing Haskell I/O code with the Object I/O code. For this purpose one can *lift* any `IO` action to a `GUI` action:

```
liftIO :: IO a -> GUI a
liftIO m = GUI (\ioSt -> m >>= \a -> return (a,ioSt))
```

### 2.2 A simple example

Let us write a GUI application that displays an up-down counter: a displayed number, together with a button to increment it and another to decrement it:




One writes a program with a graphical user interface by defining a value of type `GUI ()` and then "running" it by applying `startGUI`:

```
main :: IO ()
main = startGUI upDownGUI

upDownGUI :: GUI ()
upDownGUI = do { counter <- newCounter
               ; openDialog (Dialog "Counter" counter []) }

newCounter::GUI(TupLS TextControl (TupLS ButtonControl ButtonControl))
newCounter = ...to be defined shortly...
```

Here, `newCounter` creates one instance of our up-down counter, while `openDialog` opens a window in which the up-down counter is wrapped:

```
startGUI       :: GUI () -> IO ()
class Dialogs d where
    openDialog :: d -> GUI ()
```

The function `openDialog` opens a dialogue window (`Dialog ...`) whose contents can include all manner of things, which is why it is overloaded. Indeed, as you can see, the type of `newCounter` expresses the fact that it returns a component composed of three sub-components.

The next thing we must do is to define `newCounter`. A new feature of the mini Haskell Object I/O library, when compared with the Clean Object I/O library is the way local state is handled. We have chosen to use mutable variables [19] to handle local *and* public state. In this approach local state can still be encapsulated in the object, and hidden from the context in which it is used, thus supporting reusable GUI objects. Here, then, is how we define `newCounter`:

```
newCounter::GUI(TupLS TextControl (TupLS ButtonControl ButtonControl))
newCounter = do { c_state <- newMVar 0
                ; disp_id <- openId

                ; let display :: TextControl
                      display = TextControl "0" [ControlId disp_id]

                      dec, inc :: ButtonControl
                      dec = ButtonControl "-" [ControlFunction down]
                      inc = ButtonControl "+" [ControlFunction up]

                      up,down :: GUI ()
                      up   = update disp_id c_state (+ 1)
                      down = update disp_id c_state (- 1)

                ; return (display :+: dec :+: inc) }

update :: Id -> MVar Int -> (Int->Int) -> GUI ()
-- Update the MVar, and display new value in control identified by Id
update d m f = do { v <- takeMVar m
                  ; let new_v = f v
                  ; putMVar m new_v
                  ; setControlText d (show new_v) }
```

`newCounter` uses the GUI monad to create (a) a mutable cell, `c_state`, that will contain the state of the counter, and (b) a unique identifier, `disp_id`, used to name the display. Then it constructs the three sub-components, `display`, `dec`, and `inc`, composes them together using `(:+:)`, and returns the result.

To achieve all this, we used the following library functions and data types:

```
newMVar  :: a -> GUI (MVar a)
takeMVar :: MVar a -> GUI (MVar a)
putMVar  :: MVar a -> a -> GUI ()

openId   :: GUI Id
setControlText :: Id -> String -> GUI ()

infixr 9 :+:
data TupLS a b        = a :+: b
```

```
data ButtonControl    = ButtonControl String [ControlAttribute]
data TextControl      = TextControl   String [ControlAttribute]
data ControlAttribute = ControlId Id
                      | ControlFunction (GUI ())
                      | ControlKeyboard (...) (...)
                                        (KeyboardState -> GUI ())
                      | ...
```

The `MVar` family allow you to create and modify a mutable cell; these operations are described in detail in [19].

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 — see the definitions for `ButtonControl` and `TextControl` given above. The `TupLS` type allows you to compose two controls to make a larger one. Notice, though, that the entire GUI component is simply a data value describing the construction of the component.

The component can be given a behaviour by embedding callbacks in the attributes of the component. In particular, the `ControlFunction` attribute of the `inc` and `dec` controls is a callback that updates the counter. This call-back is run whenever the button is clicked; simply calls `update`. The latter updates the state of the counter, and uses `setControlText` to update the display.

In order to change GUI components we need to *identify* them. That is what `disp_id :: Id` is doing. It is used by the callbacks `up` and `down` to identify the GUI component (`disp`) they want to side-effect. Indeed, `MVar`s and `Id`s play a very similar role: an `MVar` identifies a mutable location, while an `Id` identifies a mutable GUI component. Fresh, unique `Id` values are created by `openId`.

It is very useful to be able to create `Id` and `MVar` values at any place in the program (see Section 2.3). For this reason, it is convenient to overload these functions so they can be used in either the `IO` or `GUI` monad:

```
class Ids m where
    newMVar  :: a -> m (MVar a)
    takeMVar :: MVar a -> m (MVar a)
    putMVar  :: MVar a -> a -> m ()

    openId :: m Id
    ... Ids also has other methods ...
instance Ids IO
instance Ids GUI
```

### 2.3 Concurrent talk

As a second example we take the concurrent "talk" program, depicted in Figure 1(a). Text typed into the upper panel of either window should be echoed in the lower panel of the other window.

**Receivers** This application involves two concurrent "processes", and we require a channel of communication going in each direction. The following functions manipulate channels:

```
class Ids m where ...
    openRId  :: m (RId a)
asyncSend    :: RId msg -> msg -> GUI SendReport

class Receivers rdef where
    openReceiver :: rdef -> GUI ()
instance Receivers (Receiver msg)

data Receiver msg
  = Receiver (RId msg) (msg -> GUI ()) [ReceiverAttribute]
```

A new channel is created by `openRId`, which is overloaded like `openId`, and returns a typed receiver name of type `RId`. You can send a message to a receiver using `asyncSend`. That triggers a callback in a (non-displayed) component of type `Receiver`. The latter contains its identifier together with the callback to be run when the message is received.

**Interactive processes** So the main program looks like this:

```
main :: IO ()
main = do { a     <- openRId
          ; b     <- openRId
          ; let talkA = talk "A" (a,b)
                talkB = talk "B" (b,a)
          ; startProcesses [talkA,talkB] }

talk :: String -> (RId TalkMsg, RId TalkMsg) -> Process
talk str (me,you) = Process (...) (talkGuts str (me,you))
                                  [ProcessClose (quit you)]

talkGuts :: String -> (RId TalkMsg, RId TalkMsg) -> GUI ()
talkGuts = ...to be defined...

quit :: RId TalkMsg -> GUI ()
quit = ... to be defined ...
```

The overloaded function `startProcesses` takes an interactive process definition[1] and evaluates them until all child processes have terminated. The two talk processes are identical except for their (string) name and receiver identification. This is expressed conveniently by parameterisation of the `talk` function.

We consider an *interactive process* to be a collection of GUI objects that share some common user interface. A process performs no independent computational activity other than the callback mechanism. Interactive processes are specified in the same way as all other GUI objects by means of an algebraic type constructor, which is defined in the library as follows:

```
data Process = Process (...) (GUI ()) [ProcessAttribute]
startProcesses :: [Process] -> IO ()
closeProcess   :: GUI ()
```

[1] or, in the real library, a (nested) list of them

`Process` has two mandatory arguments (we ignore the first) and a list of optional attributes. The `(GUI ())` argument is the *initialisation action* of an interactive process: it is the first action of the interactive process, and is run when the process is started by `startProcesses`. We will consider only one process attribute, `ProcessClose`, which is analogous to the `WindowClose` attribute discussed above: the callback associated with this attribute is evaluated whenever the user dismisses the interactive process.

In this example, the initialisation action is defined by `talkGuts`:

```
talkGuts :: String -> (RId TalkMsg, RId TalkMsg) -> GUI ()
talkGuts str (me,you)
 = do { outId <- openId
      ; inId  <- openId
      ; let talkdialog :: Dialog (TupLS EditControl EditControl)
            talkdialog = mkTalkDialog you inId outId

            receiver :: Receiver TalkMsg
            receiver   = Receiver me (receive outId) []
      ; openDialog talkdialog
      ; openReceiver receiver }

mkTalkDialog :: RId TalkMsg -> Id -> Id
                            -> Dialog (TupLS EditControl EditControl)
mkTalkDialog you inId outId
 = Dialog ("Talk "++name) (infield:+:outfield) [WindowClose(quit you)]
 where
    infield  = EditControl "" (ContentWidth "mmmmmmmmmm") 5
               [ ControlId inId, ControlKeyboard (...) (...) input ]
    outfield = EditControl "" (ContentWidth "mmmmmmmmmm") 5
               [ ControlId outId, ControlPos (Below inId,zero) ]

input :: KeyboardState -> GUI ()
input = ...to be defined...
```

This code creates two new `Id`s to identify the two panels of the window, constructs the dialogue and receiver, and then opens them. The `receiver` is straightforward — we defined `Receiver` in the previous section — and is passed the callback `(receive outId)`. The dialogue is built in a very similar way that we built the counter earlier, except that it uses editable-text panels (`EditControl`) instead of buttons. The `ControlKeyboard` attribute of the `infield` takes a callback, `input`, which tells the control how to respond to user input.

**Message passing** The remaining pieces of the puzzle are those that send messages. First we need to define the type of messages that flow between the two processes. As specified informally in the introduction, keyboard input in the input field of the talk dialogue of one interactive process should be sent to the other interactive process (and displayed in the output field). In addition, if the user dismisses either dialogue, the other one should also be notified as terminated. This is arranged by the following simple message type:

```
data Message = NewLine String | Quit.
```

The receiver callback action straightforwardly implements the informal specification above: the response to a `(NewLine text)` message should be to change the content of the output field to `text` (using the library function `setControlText`), and the response to a `Quit` message should be to terminate its parent process (using the library function `closeProcess`):

```
receive :: Message -> GUI ()
receive (NewLine text) = setControlText outId text
receive Quit           = closeProcess
```

The behaviour of the `input` callback is to read the current content of the input control and send it to the other interactive process. (The library function `getParentWindow` returns an abstract value that represent the current state of the complete dialogue. The function `getControlText` retrieves the content of any text related control. Their types are included below.)

```
input :: KeyboardState -> GUI ()
input _
    = do { Just window <- getParentWindow inId
         ; let text     = fromJust (snd (getControlText inId window))
         ; error       <- asyncSend you (NewLine text)
         ; return () }
-- Library types:
getParentWindow :: Id -> GUI (Maybe WState)
getControlText  :: Id -> WState -> (Bool,Maybe String)
```

Finally, the `quit` callback closes its own process, and sends a `Quit` message to the other process:

```
quit :: RId TalkMsg -> GUI ()
quit you = do { asyncSend you Quit;  closeProcess }
```

It should be observed that it is not possible in the Object I/O library for one interactive process to terminate other interactive processes. The `closeProcess` function has no process identification, but always terminates the interactive process of the GUI component which callback evaluates this function. This is also the case for all other actions: one interactive process can not directly create or close a window in another interactive process. The only interaction between interactive processes is message passing or via the external world.

## 3 The pros and cons of `MVars`

The mini Haskell Object I/O API discussed so far relies on mutable variables to keep track of the state of an interactive program. This is different from the way state is handled in the Clean Object I/O library. We will not discuss this system in detail, because this has been done extensively elsewhere [4,6]. Briefly, the Clean Object I/O system keeps track of all state. Type security is obtained by parameterisation of all GUI type constructors with the types of the local and public state. Specialised type constructor combinators are required to obtain the proper state encapsulation. Both approaches to handle state have been implemented in the mini Haskell Object I/O port. In this section we analyse the features of the two approaches.

Handling state with mutable variables has a number of advantages when compared with the Clean scheme.

Firstly, the set of type constructors is *simpler* (no state type variables) and *smaller* (local state type constructor combinators are superfluous). In our experience these elements of the Object I/O library cause a steep learning curve to novice GUI programmers. Despite the reduction of complexity, GUI definitions are identical to the local state version (which is identical to the Clean version). This allows to easily convert code between these versions.

The second advantage is *increased flexibility*: GUI components can share state in a more complex way than is possible in the local state version. It is unclear if it is possible to extend the local state version with more powerful state combinators, but even so this will increase the complexity of the system.

Thirdly, even though mutable variables are globally accessible, the fact that one requires its reference gives the programmer fine grained control over the actual access to the data. This is in fact analogous to the current situation with identification values: we can use the same lexical scoping techniques to control access to GUI objects as well as state objects (and get similar 'preambles' as discussed at the end of Section 2.3).

The major disadvantage of handling state with mutable variables is that it is *less declarative*. The burden of state management is shifted from the library implementer to the application programmer. To illustrate this case, here is the local state version of the up-down counter (Section 2.2):

```
newCounter
  = do { disp_id <- openId
       ; let ... control definitions are identical ...
             up   = update disp_id (+ 1)
             down = update disp_id (- 1)
       ; return (NewLS 0 (display :+: dec :+: inc)) }

update :: Id -> (Int->Int) (Int,ps) -> GUI ps (Int,ps)
update d f (v,state) = do { let new_v = f v
                          ; setControlText d (show new_v)
                          ; return (new_v,state) }
```

Instead of retrieving and storing the current count explicitly from a mutable variable, `update` has direct access to the local state, and is required by the type system to return a new value. The relation between the initial local state value `0` and the counter is determined by the `NewLS` type constructor combinator.

We need more experience to decide which of the approaches to handle state is the best choice.

## 4 The OS independent layer

The only crucial difference between the Clean and Haskell version of the Object I/O libary is the way callbacks are handled (*functions* versus *actions*). The Clean Object I/O library is a sequential implementation that encodes the interactive process scheduling mechanism and message passing. This implies that it is in principle sufficient to reimplement only the callback evaluation mechanism, and simply translate the other parts from Clean to Haskell.

In the introduction we have stated that we were going to use GHC (for issues related to other Haskell compilers we refer to Section 8). The major motivation is that the Object I/O library (in fact, some of its predecessor versions [3]) has been designed with concurrency in mind: it should, in principle, be possible to implement interactive processes as *concurrent* evaluation processes, and even to create *distributed* interactive applications. These are things that are well supported by GHC, and the combination with other required features fixed the choice for this particular compiler technology.

We are not going to discuss every detail of the implementation of the OS independent layer. Instead we focus on the following aspects of its implementation: in Section 4.1 we discuss how monadic callbacks with local state can be evaluated, in Section 4.2 we show how interactive processes are mapped to concurrent threads, and in Section 4.3 how message passing is handled.

### 4.1 Evaluation of monads with local state

Computing state transitions is straightforward in the mini Haskell Object I/O system based on mutable variables: whenever a callback action must be evaluated, the run-time system only needs to locate the proper action and apply it. All state handling is done by the callback action. In the local state version things are more complicated: the run-time system not only needs to locate the proper action, but also construct the proper state argument to apply the action to. The thus computed new state must then be restored in the administration.

The Clean Object I/O implementation uses an elegant solution to compute and store local state [5]. It relies on *lazy evaluation* to create references to local state values that will eventually be computed by callback functions (ensured by the type system). These references are stored in the internal administration (the `IOSt` environment) which is passed as the argument to the callback function. Because the involved environments are explicitly available in the Clean Object I/O library, it is rather intuitive to 'connect' these forward references. In the local state version of the mini Haskell Object I/O library we have been able to copy this strategy, using the monadic extension `fixIO`. Due to lack of space we omit a detailed presentation. Here the key idea is that `fixIO` also allows us to manipulate results that are not yet computed, but lazily available.

### 4.2 Interactive processes

The concurrent talk test case spawns two interactive processes. Interactive processes have been designed with concurrent evaluation in mind. They should behave as if they were independent applications running in a pre-emptive OS. We have implemented interactive processes using the Concurrent Haskell primitives [19] `forkIO` (for thread creation), `MVars` (for sharing context information), and `Channels` (for abstract event dispatching). Because the Microsoft Windows OS expects a single event loop driven application, we can't implement each of these processes as independent loops fetching and dispatching OS events. The architecture of the concurrent implementation is sketched in Figure 2.

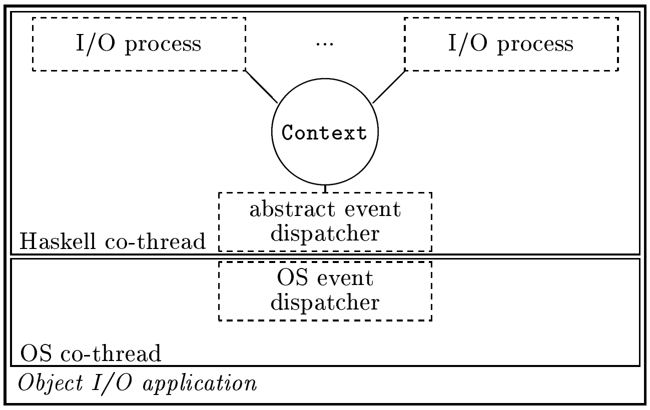



**Fig. 2.** Concurrent implementation of interactive processes

The co-thread architecture is a legacy from the Clean implementation. We can't call Clean from C. This is necessary on the Microsoft Windows platform, because some OS calls require further callbacks to be evaluated before the call is finished (for instance, when creating a window several dozens of callbacks are triggered). Instead of calling Clean from C directly, this communication is done indirectly via two OS threads that run as co-routines. Information is passed via a small globally accessible buffer. It should be noted that in Haskell one can call Haskell from C [12], so we should be able to eliminate these co-threads. Because of the preliminary nature of this project, we have not changed the code of the OS co-thread nor the architecture.

The Haskell co-thread is basically a reorder of existing pieces of functional code in the scheduling module of the Clean implementation. Because we use the `forkIO` primitive, the scheduling code disappears. Each interactive process (I/O process) runs in a Haskell thread. They are driven by an event loop that handles only abstract events. The `closeProcess` function terminates the loop, and the I/O process gets garbage collected. Abstract events are generated by an additional Haskell thread, the *abstract event dispatcher*, that maps OS events to abstract events and dispatches them to the proper I/O processes. Recall that interactive programs are created with the `startProcesses` library function. This function creates the initial Haskell threads.

The only information required by the abstract event dispatcher is which I/O processes are currently present in order to dispatch the proper abstract events. This is stored in the globally accessible `Context`, stored conveniently in a `MVar`. Every I/O process maintains an OS event filter function in this `Context`. After every callback evaluation, it updates this entry, and in the act of termination it removes it from the administration. The abstract event dispatcher terminates when this list is empty, resulting in the required behaviour of `startProcesses`.

The major advantage of this architecture is that it is scaleable: it is easy to create and destroy interactive processes. It is also very suitable for a *distributed environment*: if ever Object I/O applications distribute themselves over a network almost everything we need to do is to create a new remote initial context, an event dispatcher, and an initial I/O process thread.

### 4.3 Message passing

In the concurrent implementation as described above `Chan`nels are the obvious Concurrent Haskell medium to implement message passing. Recall that a receiver that handles messages of some type `msg` is unambiguously identified by a receiver identification value of type `(RId msg)` (Section 2.3). Its implementation is

```
data RId msg = RId { rid::Int, ridIn::(Chan msg) }
```

The `rid` field is a fresh value to uniquely identify the receiver (inherited from the original implementation). The `ridIn` field is new. It is a `Chan`nel that implements the message queue.

Messages are sent with the function `asyncSend :: RId msg -> msg -> GUI SendReport`. After the usual correctness checks it places the message in the message queue using `writeChan`. The receiver is notified that a message is available in the message queue by inserting a pseudo OS event in the abstract event stream environment (which is part of the shared `Context`). This pseudo OS event is mapped to an abstract event which is dispatched to the parent I/O process of the receiver. The receiver will eventually remove the message from its message queue (using `readChan`) and handle the appropriate callback action.

## 5 The OS dependent layer

In this project we have reused the existing C code completely. So we have integrated these C modules into the Haskell implementation. For Haskell 98 the *Foreign Function Interface* [11] has been proposed to be able to write down Haskell code that calls upon foreign functionality. We illustrate how this has been done by means of the following C procedure, defined in the module `cpicture.c`:

```
extern void WinGetStringWidth
               (CLEAN_STRING,CLEAN_STRING,int,int,int,HDC,OS,int*,OS*)
```

`CLEAN_STRING`s point to structs of a length field (`int`) and a buffer of `char`s of the given length. The types `HDC` and `OS` are also `int`egers. If a Clean function returns a tuple of results, then these are passed by the C procedure by means of pointer values (`int*` and `OS*` respectively). The C procedure returns `void`. If a Clean function returns one value then the C procedure also returns that value. The Clean code looks as follows (module `pictCCall_12.icl`):

```
WinGetStringWidth :: !{#Char} !Fnt !Int !HDC !*OS -> (!Int,!*OS)
WinGetStringWidth _ _ _ _ _
 = code { .inline WinGetStringWidth
              ccall WinGetStringWidth "SSIIIII-II"
          .end
        }
```

In the Haskell implementation we need to add marshalling code to convert the Haskell arguments to the arguments as required by the C procedures. For all functions we follow the same scheme. Here is the Haskell code:

```
winGetStringWidth :: String -> Fnt -> Int -> HDC -> IO Int
winGetStringWidth a1 (a2,a3,a4) a5 a6
    = do s1 <- createCLEAN_STRING a1
         s2 <- createCLEAN_STRING a2
         o1 <- malloc 4
         o2 <- malloc 4
         cWinGetStringWidth s1 s2 a3 a4 a5 a6 osNewToolbox o1 o2
         mapM_ freeCLEAN_STRING [s1,s2]
         r1 <- fpeek o1
         r2 <- free o2
         return r1
foreign import stdcall "cpicture" "WinGetStringWidth"
   cWinGetStringWidth :: Addr -> Addr -> Int -> Int -> Int -> HDC -> Int
                      -> Addr -> Addr -> IO ()
```

Haskell `String`s are lists of characters. These are converted to `CLEAN_STRING`s using the function `createCLEAN_STRING`. This function has been implemented in Haskell, using the GHC language extensions modules `Addr`, `Bits`, and `Storable`. For all output arguments memory is allocated, using `malloc`. This is probably extremely inefficient but at the time of writing our prior interest was correctness.

When all arguments have been created the C procedure can be called. The connection is made with the `foreign import` statement which identifies the C module and procedure name. As you can see, the type of the function closely follows that of the C procedure given above. After evaluation, the necessary results need to be freed. This is done by `freeCLEAN_STRING` for strings, and `fpeek` which before `free`ing its argument `peek`s it and returns the value.

## 6 Experience

This project was carried out by an experienced Clean programmer and an experienced Haskell programmer. This was a good occasion to compare the two languages. Clean and Haskell are clearly cousins. Proficient Clean (Haskell) programmers can master Haskell (Clean) easily. Still, the two languages have their advantages when compared with each other. Clean languages advantages are:

**Records:** Haskell field labels are fairly equivalent to Clean records. Haskell field labels automatically convert to field selector *functions*. Therefore one can't existentially quantify these fields because the field selector functions become ill-typed. In Clean, a record is basically an algebraic data type with one data constructor. The fields identify the arguments of the constructor. The normal type rules apply, including those for existential types. Finally, Clean allows the same record field name to occur in several record types. These expressions can always be disambiguated by either a unique combination of field names or by adding the type constructor name inside the record notation. As an illustration, the following definitions are valid Clean (in Haskell syntax – note the absence of a data constructor):

```
data R1 = {a::Int, b::Bool}
data R2 = {a::Bool,c::Real}
                          -- Inferred types:
f1 {a,b}     = (a,b)  -- f1 :: R1 -> (Int,Bool)
f2 {a,c}     = (a,c)  -- f2 :: R2 -> (Bool,Real)
f3 {R1 | a} = a       -- f3 :: R1 -> Int
f4 {R2 | a} = a       -- f4 :: R2 -> Bool
```

**Macros:** although in GHC one can use the C preprocessor, one can not export macros which limits their use. Constant functions don't help because you can't use them in pattern-matches.

**Strictness annotations.** Strictness is a well-established concept in Clean. One annotates data types, function argument types, and local definitions strictly. This gives Clean programmers fine grained control over evaluation order.

**Type constructor operators.** In Clean and Haskell type and data constructors have separate name spaces. In Clean these spaces contain the same range of symbols. This allows us to give all Clean Object I/O library GUI type definitions (including `:+:`) to have identical type and data constructor names everywhere.

**Module structure.** The Clean module system distinguishes *implementation* and *definition* modules. This basically means that Clean programmers write their `.hi` files. Definition modules are not allowed to be cyclicly dependent, but implementation modules are. Compiling a project therefore involves compiling a tree of modules.

Haskell language advantages are:

**Field labels** are also selector functions. As a programmer you do not have to write your own access functions. This results in elegant code.

**Derived instances** help programmers avoid writing code that can be derived by the compiler.

**Rank 2 polymorphism.** Clean and GHC Haskell support existential types. However, the GHC makes this language feature complete by extending the type system with *rank-2 polymorphism*. This extension allows one to write higher order functions on existentially quantified data structures.

**Cyclic modules.** Because Haskell modules consist of one file with an interface header (the export list), modules must be allowed to be cyclicly dependent. This increases expressiveness when compared with Clean. It has allowed us to parameterise all GUI types uniformly with respect to the state arguments. (Unfortunately, the GHC can't deal with cyclic Haskell modules without help from the programmer who is forced to write `.hi-boot` files.)

**Monads and `do`-notation** handle the environment passing part of interactive programs. The resulting code is less cluttered with environments when compared to equivalent Clean code.

The mini Haskell Object I/O library implements only a part of the Clean Object I/O library, see the left table below. The table to its right shows this for the Haskell mini Object I/O library with local state (*LS*) and mutable variables (*MVAR*). Note that because we have used a straightforward translation from Clean to Haskell this results in virtually identical sizes of the OS independent layers. Clean does not support rank 2 polymorphism which leads to significant duplication of code that handles existentially quantified data structures. The Haskell version could take advantage of this. The Haskell OS dependent layer is about twice the size of the Clean version due to marshalling (Section 5). The shared C implementation consists of 39 `*.c`, `*.h` modules, and 13132 loc.

| Clean | #mod. (%total) | #loc (%total) |
|---|---|---|
| OS indep. | 50 (45.9%) | 8956 (30.5%) |
| OS dep. | 20 (54.0%) | 3961 (53.9%) |

| Haskell | #mod. | #loc *LS* | #loc *MVAR* |
|---|---|---|---|
| OS indep. | 54 | 8583 | 8202 |
| OS dep. | 21 | 6393 | 6393 |

When writing large applications or libraries it becomes increasingly important to have dedicated development tools. From the very start Clean versions have been released together with integrated compiler/editor environments. In contrast, the GHC is a command-line based, Unix oriented system. It does not come with any development environment. Instead, one is supposed to use make files. For programmers being used to GUI based IDEs this is a rude awakening.

A final important factor with increased application sizes is the quality of the compiler. GHC's error messages are more informative than Clean's, especially when concerned with the type system. The Clean compiler is significantly faster than GHC. Compilation time measurements conducted on the Object I/O libraries indicate that Clean compiles 10 to 20 times as fast as GHC.

## 7 Related work

The main purpose of this project was to study the relationship between the Clean Object I/O library and Haskell, and to see if and how it can be implemented in a monadic framework. Except for the new approach (from a Clean point of view) to handle (local) state using `MVars`, the Object I/O library has not changed. This is the reason why we give only a very brief comparison with related work. ([5] discusses related work with respect to the 'original' Object I/O library.)

We have shown how the Object I/O library can be simplified using mutable variables. Mutable variables are used in several functional GUI systems (*TkGofer*, *pidgets*, *tclHaskell* – [26, 23, 22] respectively –). FUDGETS and *gadgets* ([10, 16]) use stream communication to model *global* state, and recursive functions for *local* state. The latter technique is used in *Opal*[13] to model *global* state. Even if (local) state is handled by means of mutable variables, the Object I/O library differs because of its emphasis on defining GUI objects by means of algebraic data structures. This allows one to define functions that pass around GUI specifications that can be manipulated and changed before they are actually created. In all other systems GUI objects can only be created, using actions.

We have demonstrated that it is possible to have a concurrent implementation of a functional GUI system without sacrificing the deterministic semantics of interactive processes. The use of Concurrent Haskell primitives results in a simpler and shorter implementation of the library, without changing its semantic properties. This contrasts strongly with the general opinion that one has to use a concurrent functional language to construct GUI programs flexibly [23, 13].

## 8 Conclusions and future work

In this project we have successfully ported an essential subset[2] of the Clean Object I/O libary to Haskell. We have argued that the Object I/O library is really independent of the underlying paradigm to integrate side effects in pure functional languages. Callbacks are modeled in Clean using explicit environment passing functions, while they are represented as monads in Haskell. In this way we preserve the best properties in both languages.

The mini Object I/O library covers 42% of the whole Object I/O library. We have focussed on the 'hard bits'. All crucial design and implementation issues have been solved. Due to lack of resources we have not been able to check uni- and bi-direction synchronous message passing and the construction of a drawing monad (which should encapsulate a `Picture` environment). Except for these parts, porting the rest of the code should not prove to be difficult.

We have used the GHC 4.08.1. It implements Haskell 98 and extends it with several language and library features. This raises the question if this project has become a GHC project rather than a Haskell 98 project. The Object I/O library can not be implemented straight away in Haskell 98. To implement the 'pure' Object I/O library one needs to extend Haskell 98 with existential types. The rest of the library can be obtained by translation from Clean to Haskell. To implement the `MVar` state version existential types are not required. The only extensions needed are `MVars`. These can be added to any Haskell compiler as a separate library, though they also require significant runtime support.

[2] Available in the GHC CVS repository at `fptools/hslibs/object-io`

We think that it is worthwhile for the Haskell and Clean community to complete this project for several reasons: *(a)* the Haskell community will obtain a GUI library that has proven itself in practice, *(b)* the library can, because of its internal architecture, be easily ported to more traditional Haskell platforms, *(c)* it will encourage code sharing between Haskell and Clean. The existence of a GUI library that is both easily portable and language independent will strengthen the position of functional languages on the long term.

## Acknowledgements

Peter would like to thank Simon Marlow, Julian Seward, and Reuben 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 [14], 189-206.

[8] 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.

[9] 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.

[10] 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.

[11] Finne, S. A Haskell foreign function interface. Available as GHC extension, see: `http://www.haskell.org/ghc/docs/latest/set/sec-foreign.html`.

[12] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. Calling hell from heaven and heaven from hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, Paris, ACM Press, pp. 114-125.

[13] Frauenstein, Th., Grieskamp, W., Pepper, P., and Südholt. Communicating Functional Agents and Their Application to Graphical User Interfaces. In [8], pp. 386-397.

[14] 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.

[15] 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.

[16] Noble, R. and Runciman, C. Gadgets: Lazy Functional Components for Graphical User Interfaces. In [14], pp. 321-340.

[17] 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.

[18] Peyton Jones, S.L. and Wadler, Ph. Imperative Functional Programming. In *Proceedings of the 20th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 10-13, 1993, 71-84.

[19] 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.

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

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

[22] Sage, M. *TclHaskell*. `http://www.dcs.gla.ac.uk/~meurig/TclHaskell/`.

[23] 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.

[24] 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.

[25] Vullinghs, T., Schulte, W., and Schwinn, Th. The Design of a Functional GUI Library Using Constructor Classes. In [8], pp. 398-408.

[26] Vullinghs, T., Tuijnman, D., and Schulte, W. Lightweight GUIs for Functional Programming. In [14], pp. 341-356.

[27] Wadler, Ph. Comprehending monads. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, Nice, 1990, ACM Press, pp. 61-78.