Interactive Functional Objects in Clean

Peter Achten and Rinus Plasmeijer

Computing Science Institute, University of Nijmegen, 1 Toernooiveld, 6525 ED, Nijmegen, The Netherlands (peter88@cs.kun.nl, rinus@cs.kun.nl)

Draft version

Abstract. The functional programming language Clean has a high level I/O system (version 0.8) in which complex yet efficient interactive programs can be created. In this paper we present its successor (version 1.0), the *object I/O system*. We consider some of the design considerations that have influenced the design of the new I/O system greatly. Key issues are *compositionality*, *orthogonality*, and *extensibility*. Apart from design, the object I/O system improves on its predecessor by two major contributions: programmers can introduce *polymorphic local state* at every (collection of) user interface component(s) and programmers can create *interactive processes* in a flexible way. All interface components can communicate with each other by sharing state but also using powerful message passing primitives in both synchronous, asynchronous, and uni- or bi-directional way. Using the message passing mechanism *remote procedure calling* can be added easily. The result is an *object oriented* I/O system. As in the previous system the uniqueness type system of Clean, offering the possibility to do destructive updates in a pure functional framework, plays a crucial role. Furthermore, the object I/O system makes extensive use of new type system facilities, namely *type constructor classes* and *existential types*.

1 Introduction

In the pure, lazy, functional programming language Clean [7] [18] [19] one can develop real world applications using a high level I/O system, released as version 0.8 [2], [3]. It is available for programmers in the form of a library. The major part of the library is written completely in Clean, and only a small part interfaces with the operating system. The whole I/O system is defined in a pure functional framework. The I/O system has continued to evolve and various new advanced features have been added. The two major contributions are the ability to introduce *polymorphic local state* at every user interface element, and to create *interactive processes* dynamically that communicate by means of *message passing*. These new features allow programmers to construct programs in an *object oriented* way. The results of these research activities can be found in [1]. While incorporating these ideas in the Clean I/O system we have further refined and changed these concepts. This labour has resulted in the new version 1.0 of the I/O system, released recently. In this paper we discuss the concepts that distinguish this new I/O system, the *object I/O system*, from its predecessor.

The old I/O system has a number of features that make it a unique solution in the range of functional I/O systems (see Section 6 for a discussion of other systems). In a nutshell these are:

- The Clean programming language employs the type system, the *uniqueness type system*, to guarantee that functions have single threaded access to arguments [21] [5] [6]. This makes it possible to deal with side-effects in a flexible way without sacrificing the purity of the language.
- Based on the uniqueness type system the I/O system of Clean can use the *world as value* paradigm. This paradigm is explained in detail in [1]. Briefly, every interactive Clean program is a function of type `*World -> *World`. `World` is an abstract type which represents the complete environment of the program. The `*` in front of a type constructor states that that type is unique, enabling the destructive update of a value of that type. The `World` itself is composed of sub worlds (which may also be unique), for instance the file system and event stream. The file system is again composed of unique sub worlds, the individual files. This paradigm has two major advantages over other paradigms:
  - Real world resources are *first class citizens*: they can be passed along explicitly by functions. There is no need to encapsulate them in one single abstraction, like for instance the monadic approach [22]. The type system guarantees that the program never duplicates unique resources.
  - Programs can access an arbitrary number of real world resources without fixing a specific order of evaluation because each of these resources can be manipulated independently, and, in principle, even in parallel.
- Graphical user interfaces are specified on a high level of abstraction. Specifications are given by defining appropriate instances of predefined *algebraic data types* and *higher order functions*. With the algebraic data types a programmer describes what user interface elements should be created by the system (for instance the definition of a menu along with its items). These definitions are parameterised with functions that define what action should occur (for instance in case a menu item is selected). These functions are the callback routines of the user interface. The only other thing the programmer needs to do is to specify an initial program state of arbitrary type which holds the data required during evaluation of the program. Because the specification elements are all values they can be interpreted easily by a special library function `StartIO` which does all the complicated and low level event handling. The semantics of a program is that of a simple *state transition system*. The state consists of the program state and an abstract data type, the `IOSt` (pronounce 'I/O state') in which the information of the concrete interface elements is maintained. The callback functions are ofcourse the state transition functions. The program terminates as soon as one of the callback functions has applied the library function `QuitIO` to the `IOSt`. `QuitIO` closes

all currently open user interface elements and returns a special empty `IOSt` value. This causes termination of `StartIO`.

– Because of the high level of abstraction the system is portable to a wide variety of very different and incompatible platforms. Currently ports exist for Macintosh, X Windows, Linux, Microsoft Windows(95 and NT), and OS/2. Each port takes care that the high level specification is mapped to its corresponding platform maintaining the required look and feel of that platform.

As an example consider the following (subset of) the algebraic data types that are used in the old I/O system to define menus:

```
::  MenuDef     s io
=   PullDownMenu Id MenuTitle            SelectState [MenuElement  s io]
::  MenuElement s io
=   MenuItem     Id ItemTitle KeyShortcut SelectState (MenuFunction s io)
|   MenuSeparator
```

A possible instance of a menu definition is `menu`, shown below together with a picture of its mapped look on a Macintosh platform.

```
menu :: MenuDef s (IOSt s)
menu
= PullDownMenu menuId "File" Able
  [ MenuItem id1 "Open..." NoKey Able   open
  , MenuItem id2 "Close"   NoKey Unable close
  , MenuSeparator
  , MenuItem id3 "Quit"    NoKey Able   quit
  ]
```

This data structure is interpreted by the I/O system which will generate a concrete menu. This menu will be a pulldown menu with the title "File". The `Able` attribute denotes that the menu is selectable. To change the menu afterwards the program uses the Id menuId. The "File" menu has four elements, three of which are commands (`MenuItem`) and one is a visual separator (`MenuSeparator`). If the user selects in one way or another the "Quit" menu item then the callback function of "Quit", `quit`, will be evaluated. The type of a callback function is in general: `s -> (IOSt s) -> (s,IOSt s)`, with `s` the program state. A typical definition of `quit` is the following:

```
quit :: s -> (IOSt s) -> (s,IOSt s)
quit programstate iostate = (programstate, QuitIO iostate)
```

For any program state and `IOSt` `quit` will terminate the application because it applies `QuitIO` to its `IOSt` argument.

The release of the old I/O system has enabled us to gather feedback from ourselves and external users. On our own part we investigated the expressive power and efficiency of the system. Two examples of large applications that we have created in Nijmegen are the Clean programming environment and a functional spreadsheet [13]. The first application is interesting because it demonstrates that the efficiency of Clean and the I/O system is that good that full screen editors and the like can be created and used in daily practice. The Clean programming environment comes with the standard distribution of the Clean system. The latter application is interesting because it consists of three independently developed components: a spreadsheet, a symbolic evaluator and small proof tool, and a text editor that have been composed afterwards into one application. A thorough early elaborate external examination is [16]. From the implementation of various ports we were able to evaluate the level of abstraction.

Not only the I/O system has been subject of reflection and feedback. The Clean language itself has also been improved, partially due to this research. The most important changes of Clean with respect to this project concerns its type system which has been extended with *record types*, *existential types* [15] and *type constructor classes* [14].

These experiences enabled us to improve the design of the I/O system considerably with respect to *compositionality*, *orthogonality*, and *extensibility*, see Section 2.

Apart from improvements on the design of the I/O system two more fundamental issues required thorough investigation:

- Interactive programs use the program state to store information they need during evaluation. For small programs this is usually fairly trivial and causes no problems, but as programs grow things get complicated quickly. Because the program state is in principle accessible by every callback function it is difficult to guarantee the integrity of data. For this reason the new I/O system should allow programmers to add *polymorphic local state* at every (collection of) interface component(s). This is discussed in Section 3.

- The functional spreadsheet case study [13] illustrates the need for a flexible set of combinators that allow programmers to merge independently developed interactive programs into a larger one. In the old I/O system this composition is limited to (arbitrarily deep) nested composition. The ultimate goal is a set of combinators in which interactive programs can be created and destroyed as independent *interactive processes*. *Message passing* takes care of the interprocess communication. This is discussed in Section 4.

Being able to encapsulate polymorphic state in the form of interactive processes and having a flexible means of message passing turns the new I/O system in an *object oriented* I/O system. This issue is surveyed in Section 5. We end this paper with a brief summary of related work in Section 6 and present conclusions and future work in Section 7.

2 Design improvement

One of the factors that helped us to improve the design of the I/O system was the addition of many new language facilities in Clean.

The introduction of record types helps a lot when programming interactive applications. Before the introduction of record types the program state was usually defined by an algebraic data type. Access to program state components was done by pattern matching. When changing the definition of the program state (a frequently occuring action while developing programs) all functions that pattern match the program state needed to be rewritten. Ofcourse, using a disciplined style of introducing (abstract) access functions from start should be used, but this also requires work. Records make the definition of functions more independent from changes of the record type.

Another change of Clean is a syntactic feature which improves the readability and comprehension of interactive functions. In the world as value paradigm functions pass around environments as unique values. Each of these values has to have a unique identifier. Therefore in a typical function definition the identifiers of these environments are usually tagged with numbers to be able to distinguish them. Here is a small example function which uses two imaginative functions `getchar` and `putchar` to implement an echo function.

```
echo :: *World -> *World
echo world
|   c=='\n'     = world2
|   otherwise   = echo world2
where
    (c,world1) = getchar world
    world2     = putchar c world1
```

In Clean it has now been made possible to introduce a new lexical scope, using the keyword `#`, inside a function definition, indicating a let-expression which can be defined before a guard. In such an expression identifiers on the right-hand-side are allowed to be reused on the left-hand-side; they are internally tagged with a number. Comprehension is aided because evaluation order corresponds with the textual order. This pays of considerably when passing around environments. Using `#` syntax `echo` can be written as:

```
echo :: *World -> *World
echo world
#   (c,world) = getchar world
    world     = putchar c world
|   c=='\n'   = world
|   otherwise = echo world
```

The design of the I/O system has been improved greatly by sticking to the following relatively simple design decisions:

- The program state and `IOSt` values are now collected in a new record type `PSt` (pronounce ‘process state’). Its type definition is:

```
::  *PSt ps
    =   {   ps :: ps
        ,   io :: *IOSt ps
        }
```

All callback functions have the same basic type, `(PSt ps) -> (PSt ps)`, the identity type on process state. This makes function composition a lot easier.

– All attributes for which sensible default values can be chosen have been made optional. For instance, for a menu item definition its title must be provided by the programmer whereas its id, selection and mark state, short key access, and even callback function are optional. Definitions become more concise and it is easier for the library designer to introduce new attributes without changing existing programs. One nasty property of the old I/O system is that every interface component needs to be identified by an Id value even if the component will never be changed. Making the Id attribute optional frees programmers from this pain.

– All interface components can be created and destroyed dynamically. This gives the programmer the ability to change the layout and content of the user interface of a program easily.

Another major improvement in the design of the I/O system concerns windows and dialogues. In the object I/O system the definitions of these ubiquitous user interface elements have been unified. Now both windows and dialogues can contain arbitrary complex and recursive controls. The ability to regard a composition of controls as a new single control, the *compound control*, increases the expressive power of the layout language considerably. Controls can be positioned relative to the frame of its parent component (window, dialogue, or compound control), but also relative to other controls or at absolute locations.

Finally, because we have gained a lot of implementation experience due to the ports to many different platforms, we have learned how to arrange the object I/O system in such a way that it anticipates differences between these platforms more easily.

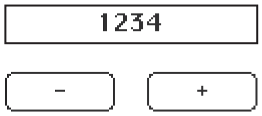
3 Polymorphic local state

In this section we present the techniques that we have applied to incorporate polymorphic local state in the object I/O system. We will first illustrate our intention by means of a small example of a counter in a system without local state. For this purpose we use the following algebraic data types to define controls:

```
:: Control ps
    = TextControl     String       [ControlAttribute ps]
    | ButtonControl   String       [ControlAttribute ps]
    | CompoundControl [Control ps] [ControlAttribute ps]
:: ControlAttribute ps
    = ControlId       Id
    | ControlPos      ItemPos
    | ControlFunction (ps -> ps)
```

The example counter consists of a display that shows the current value of an integer. This value can be decremented and incremented by the user by pressing two buttons labeled `"-"` and `"+"` respectively. Let `init` be the initial value of the counter. For conciseness we assume that the program state consists of the count value only. The counter is defined by the following expression:

```
counter :: Control (PSt Int)
counter
= CompoundControl
  [ TextControl (toString init) [ControlId dispid,   ControlPos Center]
  , ButtonControl "-"    [ControlFunction (upd (-1)),ControlPos Center]
  , ButtonControl "+"    [ControlFunction (upd 1)]
  ] []
```



The function `toString` is an overloaded function that converts its argument to a string. A text control is used to represent the integer display. It is identified by the Id value `dispid`. The text of a text control can be changed afterwards by the function `SetControlText` which expects this Id. The callback function of the decrement and increment button is defined as follows:

```
upd :: Int (PSt Int) -> (PSt Int)
upd dx {ps=count,io}
=   {ps=newcount,io=SetControlText dispid (toString newcount) io}
where
    newcount = count+dx
```

So selection of the `"-"` (`"+"`) button adds the value -1 (1) to the current count value that is located in the program state and updates the display.

For a small program using one program state for information that is maintained by a part of the program does not cause any problems. As discussed in the introduction, when programs get more complex it becomes increasingly difficult to guarantee the integrity of the program state.

What we really want is to change the I/O system in such a way that the program state contains only the *global information*, and that where needed, interface components can have their private *local state*. The I/O system has to prevent other parts of the program to have access to this local state. In case of the counter we want to associate the integer counter explicitly with the counter. Therefore, if the callback function of one of the counter controls is evaluated it should not be a function of type `(PSt Int) -> (PSt Int)` but of type `(Int,PSt ps) -> (Int,PSt ps)`. In this way we can define `upd` by:

```
upd :: Int (Int,PSt ps) -> (Int,PSt ps)
upd dx (count,ps)
=   (newcount,appPIO (SetControlText dispid (toString newcount)) ps)
where
    newcount = count+dx
```

(The library function `appPIO` updates the `io` component of a `(PSt ps)` record using an `(IOSt ps)` transition function argument.)

There are two obstacles when attempting to redefine the algebraic type definition `Control` as given in the beginning of this section to accomodate arbitrary compositions of controls with and without local state. Firstly, `Control` has only one parameter `ps` which represents the process state. We need to extend it with an alternative in which we associate a local state of type `ls` with a recursive control definition that expects a local state of type `ls` and a process state of type `ps`. This can only be a pair, and so the type of the recursive controls will be `Control (ls,ps)`. This continues recursively for each additional local state. This is ofcourse not the kind of compositional behaviour that we are looking for. For instance, one can not add controls without local state to such recursive controls. Secondly, lists can not be used to construct recursive collections of controls with and without local state because their elements have to have the same type.

More powerful glue is required to construct a set of types in which interface elements can be composed. This glue is provided by Clean *type constructor classes*, similar to Gofer [14]. The type constructor classes restrict what elements can be glued, while their instances define what the elements are. So we first start to define the instance elements. This is done by promoting the alternatives of the set of algebraic types to individual type constructors. At positions where lists are used to create recursive elements a type constructor variable is introduced. So, the algebraic type `Control` above is changed into:

```
::  TextControl       ps = TextControl     String [ControlAttribute ps]
::  ButtonControl     ps = ButtonControl   String [ControlAttribute ps]
::  CompoundControl c ps = CompoundControl (c ps) [ControlAttribute ps]
```

These type constructors will define controls *without* local state. The type constructor class that creates such controls and adds them to a window is `Controls`:

```
class Controls cdef
where
    OpenControls :: Id (cdef ps) (PSt ps) -> (PSt ps)

instance Controls TextControl
instance Controls ButtonControl
instance Controls (CompoundControl cdef) | Controls cdef
```

This definition states that controls without local state are either a `Text-Control`, or a `ButtonControl`, or a `CompoundControl` of `cdef` provided that `cdef` also belongs to the type constructor class `Controls`.

From the type definitions of controls without local state we derive the type definitions of controls *with* local state. To obtain unique names, `LS` is appended at the type and data constructor names. The type constructors obtain one additional type parameter `ls` which corresponds with the local state. The attributes (which contain the callback functions) operate on the pair of local state and process state `(ls,ps)`.

```
::  TextControlLS        ls ps
=   TextControlLS      String   [ControlAttribute (ls,ps)]
```

```
::  ButtonControlLS      ls ps
=   ButtonControlLS    String   [ControlAttribute (ls,ps)]
::  CompoundControlLS  c ls ps
=   CompoundControlLS (c ls ps) [ControlAttribute (ls,ps)]
```

The type constructor class that creates controls with local state, `ControlsLS`, is very similar to `Controls`. It has one additional argument which defines an initial value for the local state.

```
class ControlsLS cdef
where
    OpenControlsLS :: Id ls (cdef ls ps) (PSt ps) -> (PSt ps)

instance ControlsLS TextControlLS
instance ControlsLS ButtonControlLS
instance ControlsLS (CompoundControlLS cdef) | ControlsLS cdef
```

This definition states that controls with local state are either a `TextControlLS`, or a `ButtonControlLS`, or a `CompoundControl` of `cdef` provided that `cdef` also belongs to the type constructor class `ControlsLS`.

To glue elements that have the same kind two new combinators are predefined, one for elements without local state (`:+:`), and one for elements with local state (`:~:`). Their definition is:

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

For instance, using `:+:` and the new type constructors for controls without local state we can rewrite the original definition of counter as follows:

```
counter
= CompoundControl
  (   TextControl (toString init) [ControlId dispid,ControlPos Center]
  :+: ButtonControl "-" [ControlFunction (upd (-1)),ControlPos Center]
  :+: ButtonControl "+" [ControlFunction (upd 1)]
  )   []
```

This definition not only differs in the use of the `:+:` constructor instead of list constructors, it also has a more elaborate type than its original, concise, type `Control (PSt Int)`:

```
counter :: CompoundControl
           ((:+:) TextControl
           ((:+:) ButtonControl
                  ButtonControl
           ))     ps
```

It is interesting to observe the structural equivalence between the type of `counter` and its definition. In our approach the types already contain a lot of information about their 'inhabitants'. Although in this scheme types grow proportionally with the number of type constructors involved in the definition, programmers do not have to worry because the Clean compiler can derive these types automatically.

Using the new glue we are can compose elements with local state or without local state. To allow us to switch between these realms the object I/O system has four type constructor combinators. These combinators either *introduce* a local state or *eliminate* a local state. Switching from an element without local state to an element with local state can be done easily by introducing a type variable. However, switching from an element with local state to an element without local state encapsulates a concrete local state value. In other words, in the type definition of such a type constructor combinator a type variable has to be introduced on the right hand side. In Clean this can be done by *existentially quantifying* these type variables. One (or more) type variable(s) can be introduced by existential quantification by placing them behind the keyword `E.` which appears immediately before the rest of the type.

```
::  LS       t ls ps = LS (t ps)
::  NoLS     t    ps = E.ls : {introLS ::ls, introDef ::t  ls      ps}
::  ExtendLS t ls ps = E.ls1: {extendLS::ls1,extendDef::t (ls1,ls) ps}
::  ChangeLS t ls ps = E.ls1: {changeLS::ls1,changeDef::t  ls1     ps}
```

The type constructor combinators should be read as follows:

- To every element definition with local state we can add an element definition `e` without local state by `(LS e)`.
- To every element definition without local state we can add an element definition `e` with local state of value `x` by {`introLS=x,introDef=e`}.
- To every element definition with local state we can add an element definition `e` with an extended local state of value `x` by {`extendLS=x,extendDef=e`}.
- To every element definition with local state we can add an element definition `e` with other local state of value `x` by {`changeLS=x,changeDef=e`}.

We can now construct the counter with local state:

```
counter
= { introLS = init
  , introDef
    = CompoundControlLS
       (   TextControlLS (toString init)
                             [ControlPos Center,ControlId dispid]
       :~: ButtonControlLS "-" [ControlPos Center
                             ,ControlFunction (upd (-1))        ]
       :~: ButtonControlLS "+" [ControlFunction (upd 1)           ]
       )   []
  }
```

The type of `counter` effectively hides the type of the local integer state `init`:

```
counter :: NoLS (      CompoundControlLS
                ((:~:) TextControlLS
```

```
((:~:) ButtonControlLS
       ButtonControlLS
)))    (PSt ps)
```

4 Interactive processes

In this section we present the tools that are offered in the object I/O system to create and handle *interactive processes*. To illustrate the concepts we will construct a small interactive *talk* program. To clarify the account, we start with a simplified version of the object I/O system.

The example talk program creates two interactive processes. Each interactive process has a simple dialogue with four elements:

- An input field in which the text is typed that has to be *sent* to the other talk process.
- An output field in which the most recently *received* text from the other talk process is displayed.
- A button that, when selected, sends the current input text to the other talk process.
- A button that, when selected, terminates the program.

As this example suggests, when dealing with processes, one also needs to communicate. The object I/O system provides functions to do *message passing*. Messages can be arbitrary values (data structures and functions). Messages are received by *receivers* which can be opened and closed analogously to menus, windows, and dialogues.

Basic message passing will be explained step by step as we construct our example program (see Section 4.2 for more details). For the time being we assume that we have some function `Send :: (RId m) -> m -> (PSt ps) -> PSt ps`. The abstract type `RId m` uniquely identifies a receiver that accepts messages of type `m`. When applied to such a special identifier, and a message of type `m`, `Send` will send the message to the indicated receiver.

Interactive processes are defined in the same way as user interface components in the previous section. We again add some simplifications (see Section 4.1 for more details). Algebraic data types are applied to define what interactive processes we want to open. An interactive process is a state transition system. So one component of a process definition should be the initial program state, of type `ps`. Initially, an interactive process has an empty user interface. It is up to the *process initialisation* attribute, of type `[(PSt ps) -> (PSt ps)]`, to create the user interface. The functions in the initialisation attribute are evaluated in order of appearence. The algebraic type that defines an interactive process can therefore be defined as follows:

```
::  Process
    =   E.ps:Process ps (ProcessInit (PSt ps))
::  ProcessInit ps
    :== [ps -> ps]
```

The talk program consists of two identical interactive processes. So they can be defined by the same function `talk`. The messages will either be a line of text (a `String`) or a message to request termination. This is given with the algebraic data type `Message = Line String | Bye`. For each `talk` process `me` identifies its own receiver, while `you` identifies the receiver of the other `talk` process, both of type `RId Message`. Because talk is very simple it will have no need of global nor local data, so the initial program state can be any arbitrary value. Since we have to choose a value we will use the simple type `Nil = Nil`. The initialisation actions of a talk process first creates the dialogue described above and then a receiver.

```
talk :: (RId Message) (RId Message) -> Process
talk me you
=  Process Nil [ OpenDialog   (talkdialog   you)
               , OpenReceiver (talkreceiver me )
               ]
```

To define the talk dialogue we use the controls class that we defined at the end of Section 3. For the two buttons we can conveniently use the `ButtonControl`s. The input and output controls will be represented with a library control that we haven't introduced yet, the `EditControl`. An `EditControl` can be used to display an arbitrary large text within a frame of a certain width and a fixed number of lines. Its library definition without and with local state is:

```
::  EditControl      ps
=   EditControl   String Width NrLines [ControlAttribute     ps ]
::  EditControlLS ls ps
=   EditControlLS String Width NrLines [ControlAttribute (ls,ps)]
```

So we use the `EditControl` for both the input and the output control of the talk dialogue. To prevent users from typing text in the output control, its `SelectState` attribute is set to `Unable`. The Ids `inId` and `outId` identify the input control and the output control respectively. In the I/O system controls must always be part of a window or dialogue. The talk dialog can be defined as follows:

```
talkdialog you
= Window "Talk"
    (   EditControl "" 200 5 [ControlId inId,      ControlPos Center]
    :+: EditControl "" 200 5 [ControlId outId,     ControlPos Center
                             ,ControlSelectState   Unable           ]
    :+: ButtonControl "Send" [ControlFunction (talksend you)
                             ,ControlPos Center                     ]
    :+: ButtonControl "Quit" [ControlFunction (talkquit you)        ]
    )   []
```

In the object I/O system receivers are user interface elements that handle messages of some type `m`. The data types that define receivers are (again, without and with local state):

```
::  Receiver   m    ps = Receiver   (RId m) (ReceiverFunction m     ps)
                                            [ReceiverAttribute     ps ]
::  ReceiverLS m ls ps = ReceiverLS (RId m) (ReceiverFunction m (ls,ps))
                                            [ReceiverAttribute  (ls,ps)]
::  ReceiverFunction m ps
    :== m -> ps -> ps
```

The event handler of `talkreceiver`, `talkreceive`, has a straightforward definition. If the message is `Line line`, it should set the text of the output control to `line`. If the message is `Bye`, it should terminate its parent process. The library function `CloseProcess` is the new name of the `QuitIO` function of the old I/O system.

```
talkreceiver :: (RId Message) -> Receiver Message (PSt ps)
talkreceiver me = Receiver me talkreceive []

talkreceive :: Message -> (PSt ps) -> (PSt ps)
talkreceive (Line line) ps = appPIO (SetControlText outId line) ps
talkreceive Bye         ps = appPIO CloseProcess ps
```

The only parts that remain to be defined are the two callback functions of the button controls.

The callback function of "Send", when selected, first retrieves the current content, `line`, from the input control (using the library function `GetControlText`). It then resets the content of the input control to the empty string, and sends the `Line line` message to the other talk process. (The library function `accPIO` accesses the `io` component of a `(PSt ps)` record using an `(IOSt ps)` access function argument.)

```
talksend :: (RId Message) -> (PSt ps) -> (PSt ps)
talksend you ps
#   (line,ps) = accPIO (GetControlText inId)    ps
    ps        = appPIO (SetControlText inId "") ps
=   Send you (Line line) ps
```

The callback function of "Quit", when selected, sends the `Bye` message to the other talk process and terminates the interactive process.

```
talkquit :: (RId Message) -> (PSt ps) -> (PSt ps)
talkquit you ps = appPIO CloseProcess (Send you Bye ps)
```

This finishes the definition of a talk process. All we need to do is to actually open two process instances. The function `StartProcesses` takes a list of process definitions and creates the processes by applying their initialisation functions. `StartProcesses` is the new name of the `StartIO` function of the old I/O system. `StartProcesses` terminates only if there are no more interactive processes to evaluate. Receiver identification values are generated by the overloaded function `OpenRId` for which `World` and `(IOSt ps)` are instances. So the main function of the Clean program that creates the talk application can be defined as follows:

```
Start :: *World -> *World
Start world
```

```
#   (a,world) = OpenRId world
    (b,world) = OpenRId world
=   StartProcesses [talk a b, talk b a] world
```

4.1 More about interactive processes

In the `talk` example we have used a simplified version of the set of functions and type definitions by which one can define and create interactive processes in the object I/O system. In this section we will discuss the system in more detail.

We have seen how to construct an application that consists of two independent interactive processes that communicate by means of message passing. This is not the only way to construct programs. Consider for instance the situation in which one has created an interactive process that implements a structural text editor for some programming language, and a compiling interactive process for that source language. When glueing these processes into a programming environment application it is natural that they *share* the program text that has to be edited and compiled. Another example of state sharing is the *clipboard* that is shared by applications on graphical user interface platforms.

The object I/O system allows interactive processes to share the program state, provided ofcourse that the types of the program states are equal. A collection of interactive processes that share a program state is called a *process group*. However, the restriction to equal types of program state is not a practical one. One would rather have a part of the program state to have equal type and another part to have different, arbitrary type. This is done by a small change in the process state record type definition:

```
::  *PSt ls ps
    =   {   ls :: ls
        ,   ps :: ps
        ,   io :: *IOSt ls ps
        }
```

`IOSt` is also changed in this way. The function that creates a number of shared interactive processes is the following:

```
class ShareProcesses pdef :: (pdef ps) (PSt ls ps) -> PSt ls ps
```

The instances of `ShareProcesses` are either individual interactive process definitions or interactive process compositions, using the type constructor combinators `ListNoLS` (new) and `:~:` (already introduced). The type constructor combinator `ListNoLS` is defined by `ListNoLS t ps = ListNoLS [t ps]`. It allows the convenient definition of interactive processes of the same type. Because the program state is split, the definition of an individual interactive process also has a small change. For a given shared program state `ps`, each interactive process only needs to introduce a local program state `ls`:

```
::  Process ps
    =   E.ls:Process ls (ProcessInit (PSt ls ps))
```

Expressed in Clean, the instances of `ShareProcesses` are therefore:

```
instance ShareProcesses Process
instance ShareProcesses (ListNoLS  pdef)  | ShareProcesses pdef
instance ShareProcesses (:+: pdef1 pdef2) | ShareProcesses pdef1
                                          & ShareProcesses pdef2
```

In the `talk` example we have used the function `StartProcesses` to actually create the two talk processes. With `StartProcesses` one creates an initial interactive process structure.

```
class StartProcesses pdef :: pdef *World -> *World
```

The instances of `StartProcesses` are either individual process groups or process group compositions, using lists (as we have seen in the `talk` example) and the new type constructor combinator `:^:`. The latter can be used to glue arbitrary expressions of arbitrary type:

```
::  :^: t1 t2 = (:^:) infixr 9 t1 t2
```

A process group is defined using the algebraic type `ProcessGroup`. Its definition consists of a value for the shared program state component of type `ps`, and some composition `pdef` of interactive processes that share `ps`. Its type definition is:

```
::  ProcessGroup pdef
    =   E.ps: ProcessGroup ps (pdef ps)
```

Ofcourse, one also wants to create process groups within an interactive process. This is done with the function `OpenProcesses`. It has the same type as `StartProcesses` except that the `World` argument and result should be of type `(PSt ls ps)`. Therefore `StartProcesses` and `OpenProcesses` are member of the class `Processes`. Its definition and instances are:

```
class Processes pdef
where
    StartProcesses :: pdef *World      -> *World
    OpenProcesses  :: pdef (PSt ls ps) -> PSt ls ps

instance Processes (ProcessGroup pdef) | ShareProcesses pdef
instance Processes [pdef]              | Processes pdef
instance Processes (:^: pdef1 pdef2)   | Processes pdef1
                                       & Processes pdef2
```

4.2 More about message passing

In the construction of the `talk` example, we used a function `Send` to send messages. In this section we present the means to do message passing in the object I/O system. Messages can be sent either *asynchronous* or *synchronous*, but also *uni-directional* or *bi-directional*. Although one would assume that this results in four combinations, there are only three. The combination *asynchronous/bi-directional* is not provided because such a function can be defined in terms of two asynchronous/uni-directional message passing calls making use of the fact that receivers can be created and destroyed dynamically.

We start our discussion with the simplest of the three message passing functions, the combination *asynchronous/uni-directional*:

```
ASyncSend :: (RId m) m (PSt ls ps) -> (SendReport,PSt ls ps)
```

`ASyncSend`, when applied to the identification of a receiver that accepts only messages of type `m` and a message of that type, simply adds the message to the message queue of the indicated receiver. The location of the receiver is immaterial, it may well be part of the same interactive process but also of another interactive process. However, it is possible that some exceptional situation occurs. For this purpose all message passing functions return a `SendReport` which gives information about the action.

```
::  SendReport
     =   SendOk
     |   SendUnknownReceiver
     |   SendUnableReceiver
     |   SendDeadlock
```

The only exception in case of `ASyncSend` is `SendUnknownReceiver` in case the indicated receiver could not be found in any of the currently open interactive processes. If it was found then `SendOk` is returned. If the indicated receiver remains `Able` and is not closed before its message queue is empty it will at some point in time after `ASyncSend` retrieve the message from the top of its message queue and apply its receiver function to the message.

Synchronous/uni-directional message passing is the next message passing function of the I/O system:

```
SyncSend :: (RId m) m (PSt ls ps) -> (SendReport,PSt ls ps)
```

`SyncSend`, when applied to the identification of a receiver that accepts only messages of type `m` and a message of that type, *blocks* its parent process, locates the indicated receiver if it exists, and if the receiver is `Able` applies the receiver function to the message, and then returns to and *unblocks* its parent process.

As this description suggests, synchronous message passing is more complicated than asynchronous message passing because it involves a *context switch*. Also more exceptions may occur. `SendUnknownReceiver` is returned in the same situation as with `ASyncSend`. `SendUnableReceiver` is returned in case the receiver could be found, but is currently `Unable`. `SendDeadlock` is returned in case the receiver could be found, is currently `Able`, but has a blocked parent process that is involved in a synchronous, cyclic communication with the current process. In all exceptional cases message passing is halted, and `SyncSend` returns to the current process which becomes unblocked.

The combination *synchronous/bi-directional* is the last message passing function:

```
SyncSend2 :: (R2Id m r) m (PSt ls ps)
          -> ((SendReport,Optional r),PSt ls ps)
```

Synchronous/bi-directional message passing is the same as synchronous/uni-directional message passing except for the fact that the involved receiver accepts not only a message of type `m`, but also returns a response of type `r`. For this reason a new identification type `(R2Id m r)` for bi-directional receivers is introduced that is parameterised with both the message type `m` and the response type `r`. Also the callback function of a bi-directional receiver is now not a function of type `m -> ps -> ps`, but of type `m -> ps -> (r,ps)`. This results in the following type definitions for bi-directional receivers (as usual without and with local state):

```
::  Receiver2   m r    ps
=   Receiver2   (R2Id m r) (Receiver2Function m r     ps)
                           [ReceiverAttribute         ps ]
::  Receiver2LS m r ls ps
=   Receiver2LS (R2Id m r) (Receiver2Function m r (ls,ps))
                           [ReceiverAttribute     (ls,ps)]
::  Receiver2Function m r ps
    :== m -> ps -> (r,ps)
```

Bi-directional receiver identification values are generated by the overloaded `OpenR2Id` function for which `World` and `(IOSt ls ps)` are instances. For `Sync-Send2` the same exception results apply as for `SyncSend`. In case the communication was succesful, `SendOk` is returned, but also a response value (the `Optional` type is defined by `Optional x = One x | None`). In case of an exception, the response value is `None`.

5 Object orientation

In Section 3 we have explained how polymorphic local state can be added at will by the programmer to arbitrary interface elements. In Section 4 we added interactive processes and message passing. Using these concepts it becomes possible to construct programs in an *object oriented* way. Except for inheritance, the concept of *objects* as defined in [11] with respect to Smalltalk coincides exactly with interactive processes in the object I/O system: interactive processes have local memory (the local state), inherent processing ability (obviously), and the capability to communicate with other objects (message passing between processes). *Methods* correspond closely with bi-directional receivers in the object I/O system, while invoking a method corresponds with sending the appropriate message by means of the synchronous/bi-directional `SyncSend2` primitive. In principle, inheritance can be added to the Clean language.

In the object I/O system we can use the power of functional programming languages to abstract away the fact that interactive processes use message passing to invoke actions from each other. This approach has been explained in detail in [4]. It transforms a message passing protocol into a *remote procedure calling* protocol. This means that we can choose to use either a message passing style or a functional style.

In the object I/O system we have applied this technique to rearrange the model of the `World`. In the object I/O system all interactive processes in principle should have access to the file system. Modeling the file system as a unique sub environment of type `Files` of the `World` causes unwanted sequentialisation.

Instead, we let `Files` be contained in the program state of an interactive process, the *file server* that is always part of the `World`. The file server consists of one bi-directional receiver only. Its Id is globally known in the library but hidden from the programmer. The file system functions of the old I/O system are used locally by the file server. For the programmer opening and closing files is now done in a functional style using remote procedure calls as sketched above. This new model of the world is not only more realistic, it also allows process groups to be evaluated concurrently.

6 Related work

In the area of functional languages a lot of solutions for graphical user interface I/O have emerged, and it is beyond the scope of this paper to compare them all. Therefore we will discuss only a few characteristic exponents. FUDGETS [8] and *Gadgets* [17] are examples of stream processing I/O solutions. Elements have input streams (one for FUDGETS, many for Gadgets) and output streams (one for FUDGETS, many for Gadgets). Communication occurs by placing messages on these streams.

There is a large class of solutions that rely on monads [22] to thread one external environment. The three examples that we discuss have in common that they add concurrency to the functional language to obtain a flexible I/O model for graphical user interfaces. The example systems are *Haggis* [10], *Pidgets* [20], and *TkGofer* [9]. In Haggis a lot of attention is paid to compositionality and extensibility. Pidgets has looked at the definition of user interface elements at the functional level and extends this model with an automatic hit detection mechanism. TkGofer has used type constructor classes to structure the graphical user interface element hierarchy. It must be observed however that in these systems the a-priori use of concurrency destroys the deterministic behaviour of these systems. As an alternative to repair the effects of introducing to much concurrency the Brisk project [12] pays effort to introduce only deterministic concurrency.

7 Conclusions and future work

In this paper we have presented a pure functional I/O system that allows the definition of interactive programs on high level of abstraction. It builds on the unique characteristics of its predecessor, improving on compositionality, orthogonality, and extensibility. The capability to introduce polymorphic local state everywhere and create interactive processes that communicate by means of message passing provide the programmer with powerful tools to construct programs in an object oriented style.

Thanks to the uniqueness type system Clean can exploit the benefits of the world as value paradigm. The I/O system can be arranged in such a way that process groups can, in principle, be evaluated in parallel without loosing determinism. Shared processes inside one group have an interleaved behaviour.

One of the leads to future work is to see how we can employ our knowledge of concurrency to obtain a truly concurrent I/O system.

Acknowledgements

The authors would like to thank all users of Clean who have contributed to the design of the object I/O system with their constructive criticism. We also would like to thank Marko van Eekelen for commenting the structure of this paper.

References

[1] Achten P.M., *Interactive Functional Programs - Models, Methods, and Implementation.* PhD.Thesis, Februari 1996, University of Nijmegen.

[2] 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.

[3] 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.

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

[5] Barendsen, E. and Smetsers, J.E.W. Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract). In Shyamasundar, R.K. ed. *Proceedings of the Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, 1517 December 1993, Bombay, India. LNCS **761**, Springer-Verlag, Berlin, pp. 41-51.

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

[7] 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.

[8] 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.

[9] Claessen, K., Vullinghs, T., and Meijer, E. Structuring Graphical Paradigms in TkGofer. In *Proceedings of the 1997 ACM SIGPLAN International Conference of Functional Programming* (ICFP'97), Amsterdam, The Netherlands, 9-11 June 1997, ACM Press, pp.251-262.

[10] Finne, S. and Peyton Jones, S. Composing Haggis. In it Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics, Maastricht, The Netherlands, September 1995, Springer-Verlag.

[11] Goldberg, A. Object-Oriented Programming Languages. In Sebesta, R.W. *Concepts of programming languages* (3rd ed.). Addison-Wesley Publishing Company 1995, pp. 570-607.

[12] Holyer, I., Davies, N., and Dornan, Ch. The Brisk Project: Concurrent and Distributed Functional Systems. In *Proceedings Glasgow Workshop on Functional Programming*, 10-12 July 1995, Ullapool, Scotland.

[13] Hoon, W.A.C.A.J. de, Rutten, L.M.W.J., and Eekelen, M.C.J.D. van. Implementing a functional spreadsheet in Clean. In *Journal of Functional Programming* **5**(3) - July 1995, Cambridge University Press, pp. 383-414.

[14] Jones, M.P. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, 9-11 June 1993. ACM Press, pp. 52-61.

[15] Mitchell, J.C., and Plotkin, G.D. Abstract types have existential type. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985, pp. 37-51.

[16] Noble, R. and Runciman, C. Functional Languages and Graphical User Interfaces - a review and a case study. Department of Computer Science, University of York, England, February 3, 1994.

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

[18] 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.

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

[20] Scholz, E. PIDGETS - Unifying Pictures and Widgets in a Simple Model for Concurrent Functional GUI Programming. Report Serie B 95-13, University Berlin, October 1995.

[21] 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.

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