A Framework for Deterministically Interleaved Interactive Programs in the Functional Programming Language Clean

PETER ACHTEN AND RINUS PLASMEIJER
Computing Science Institute, University of Nijmegen,
Toernooiveld 1, 6525ED, Nijmegen, the Netherlands
(e-mail: *peter88@cs.kun.nl, rinus@cs.kun.nl)*

Abstract

In this paper we present a *functional interleaved Event I/O system*. This system is a generalization of the *Event I/O system* as incorporated into the lazy, purely functional programming language Clean. The Interleaved Event I/O system offers features that are more commonly found outside the functional scene. These features are *dynamic process creation*, and two well-known forms of inter-process communication: *asynchronous message passing*, and *data sharing*. Both forms of communication are *polymorphic* and *type-safe*. As we are working in a functional language, messages can contain higher-order functions and arbitrarily complex algebraic types. Communication by data sharing is a restricted form of communication by global data structures. Nevertheless, the new system is still completely functional because the generalization is done within the pure functional framework. The Interleaved Event I/O system has been implemented and will become part of the new release of Clean.

1 Introduction

Research in the area of functional programming languages is increasingly paying more attention to the incorporation of I/O into the functional programming paradigm, starting from an early paper by Henderson (1982) to more recent work (Dwelly, 1989; Turner, 1990; Thompson, 1990; Peyton Jones and Wadler, 1993; Carlsson and Hallgren, 1993). In this paper we present some recent results of the research conducted on the incorporation of I/O into the lazy, purely functional programming language Clean (Brus *et al*, 1987; Nöcker *et al*, 1991; Plasmeijer and van Eekelen, 1993). The I/O system of Clean, the *Event I/O system*, enables programmers to have *direct* access to the file system, and to write complex Graphical User Interface applications handling windows, menus, and dialogues, at a high level of abstraction. At start, the major part of the research has focused on basic issues such as how to incorporate I/O at all into the pure functional paradigm. Clean has a special type system called Uniqueness Typing (Smetsers *et al*, 1993; Barendsen and Smetsers, 1993) that offers the possibility to directly interface the pure functional world with the imperative world by guaranteeing single threaded use of destructible objects. How Graphical User Interfaces can be suitably programmed in such a functional language has been reported in Achten *et al* (1993) and Achten and Plasmeijer (1994).

The Event I/O system is a *one process at a time* system. At all times during evaluation of an Event I/O program there is at most one interactive process running. This is not a satisfying situation for a number of reasons: programs may want to spawn interactive processes that run at the same time with the process that spawned them, and programs cannot be composed of interactive processes thus improving on the modular structure of the program. In this paper we turn the Event I/O system into the *Interleaved Event I/O system*, a system that allows programs for *many processes at a time* and sophisticated inter-process communication. These forms of communication are *asynchronous message passing* and *data sharing*. Both forms are *type-safe* and *polymorphic*. All this has been realized in a pure functional framework such that the advantages of functional programming remain. In particular can interactive processes communicate *higher-order functions* and *arbitrary data structures* to other processes.

This paper starts with a brief introduction to Clean and the Event I/O system in section 2. Section 3 describes how the Event I/O system is turned into the *Interleaved Event I/O* system: section 3.1 explains how the Event I/O system can be changed in order to handle *many processes at a time*, section 3.2 introduces *inter-process communication* by *asynchronous message passing*, and section 3.3 introduces *inter-process communication* by *data sharing*. In section 4 we present an example. Section 5 presents related work. Conclusions are drawn in section 6, and section 7 concludes with current and future work.

2 The Clean Event I/O system

Clean (Brus *et al*, 1987; Nöcker *et al*, 1991; Plasmeijer and van Eekelen, 1993) is a lazy functional programming language based on Term Graph Rewriting (Barendregt *et al*, 1987). The programs in this paper are written in Clean 1.0 (Plasmeijer and van Eekelen, 1994, *in preparation*). Most of the language constructs used in Clean 1.0 are customary in other functional languages such as Miranda[1] and Haskell. Where appropriate, the text includes remarks on peculiarities of Clean 1.0.

Interactive Clean programs are functions of type :: *World → *World. The type World is an *environment*. An environment is an *abstract data type* that encodes the state of a *specific* part of the real world (such as the file system, individual files, menus, windows, dialogues, or timers). The *type attribute* * is the type specification that the world is *unique*. The type system of Clean guarantees that any one function applied to an object of uniquely attributed type has access to this object such that the object can be destructively updated without violating the functional semantics of the language (Smetsers *et al*, 1993; Barendsen and Smetsers, 1993).

The Event I/O system provides programs with a *hierarchy* of environments that can be used to do I/O (figure 1). From the *unique* world environment the *unique file system* environment of type *Files and the *unique event stream* environment of type *Events can be retrieved from the world environment with the function OpenWorld :: *World → (*Files, *Events). The file system environment contains the individual file environments for file I/O. The event stream environment is discussed later. These two environments can create a new unique world environment with the function of reverse type CloseWorld :: (*Files, *Events) → *World.

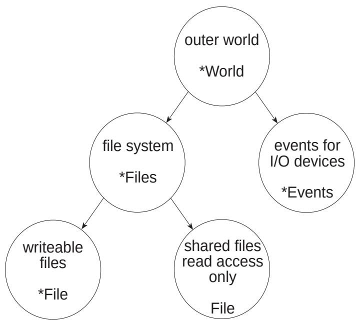


Figure 1 *The Clean environment hierarchy.*

Graphical User Interface applications are event driven. The event stream environment contains all the events that are generated at run-time by the user (using mouse and keyboard) and the operating system (for window updates). However, in Clean the programmer does not retrieve and handle events but instead uses high-level functions of the Clean I/O library that provide the programmer with an abstract view of the programming task. The Event I/O system takes care of all low level I/O handling. In order to create a Graphical User Interface application, a programmer has to:

(**a**) define the *abstract devices* to be used,

(**b**) define the *abstract event handlers* to handle the abstract events, and

(**c**) apply the predefined function StartIO to these definitions, which maps the abstract device definitions to concrete Graphical User Interface elements, and recursively handles events by calling the corresponding abstract event handlers.

(**a**) *Abstract devices* provide Clean programmers with a high level view of Graphical User Interface elements. The Clean Event I/O system has four abstract devices: the *window*, *menu*, *dialogue*, and *timer* device. Abstract interface elements are specified by functional expressions that are instances of a set of predefined algebraic types (figure 2). For each abstract device an *algebraic type* is predefined that fully specifies how the individual interface elements of that abstract device should be defined. The *ab-*

[1] Miranda is a trademark of research software Limited.

stract event handlers that are contained in the abstract device definitions are *transition functions* of type :: s → s of some *process state* of type s (see later on). Therefore the type definitions are parameterized with the type variable s.

```
::  IOSystem        s   =  [DeviceSystem s]
::  DeviceSystem    s   =  TimerSystem     [TimerDef     s]
                        |  MenuSystem      [MenuDef      s]
                        |  WindowSystem    [WindowDef    s]
                        |  DialogSystem    [DialogDef    s]
::  MenuDef s
    =  PullDownMenu      MenuId MenuTitle SelectState [MenuElement s]
::  MenuElement s
    =  MenuItem          MenuItemId ItemTitle KeyShortcut SelectState            (MenuFunction s)
    |  CheckMenuItem     MenuItemId ItemTitle KeyShortcut SelectState MarkState  (MenuFunction s)
    |  SubMenuItem       MenuId ItemTitle SelectState    [MenuElement s]
    |  MenuItemGroup     MenuItemGroupId                 [MenuElement s]
    |  MenuRadioItems    MenuItemId                      [RadioElement s]
    |  MenuSeparator
::  RadioElement s
    =  MenuRadioItem     MenuItemId ItemTitle KeyShortcut SelectState            (MenuFunction s)
::  MenuFunction s    =  s -> s
::  KeyShortcut       =  Key KeyCode | NoKey
```

Figure 2 *Cleans predefined algebraic type definitions of abstract devices and the type* MenuDef *to define individual menus. Type [α] means a list of α. The symbols printed in* **boldface** *are alternative data constructors of the algebraic type.*

As an illustration of an abstract device definition, figure 3 gives a typical example of a menu definition. The picture next to the definition shows the concrete device in the case of the menu definition being mapped to a Macintosh system.

```
PullDownMenu FileId "File"  Able [
    MenuItem NewId       "New"         (Key 'n')   Able       new,
    MenuItem OpenId      "Open…"       (Key 'o')   Able       open,
    MenuItem CloseId     "Close"       (Key 'w')   Unable     close,
    MenuSeparator,
    MenuItem SaveId      "Save"        (Key 's')   Unable     save,
    MenuItem SaveAsId    "Save As…"    NoKey       Unable     saveAs,
    MenuSeparator,
    MenuItem QuitId      "Quit"        (Key 'q')   Able       quit ]
```

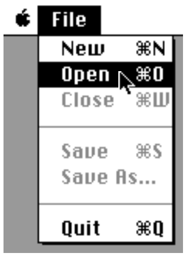


Figure 3 *An example of a menu definition in Clean.*

(**b**) *Abstract event handlers* are higher-order function arguments of abstract device definitions. They define the *response* of the interactive process to a specific *abstract event*. An abstract event is defined in the context of an abstract device. Consider for example the menu definition in figure 3. One abstract event defined in the context of this definition is *the menu item named 'Open…' has been selected*. The abstract event handler that corresponds with this abstract event is the function open.

Abstract event handlers are functions that change the *process state* of the interactive process. The process state is a predefined parameterized *record type*. In Clean a *record type* is an algebraic type with *exactly one* alternative constructor. The alternative constructor does not need to be specified if its field names uniquely identify the record type. Record types and record expressions always appear between {} in a program. The process state record type State ps consists of three fields: the *program state* of type ps which reflects the logical state of the interactive process, the unique file system environment of type *Files, and the unique *IOState, an environment that contains the run-time states of the concrete devices the interactive process uses, together with the event stream environment.

```
::  State ps = {   pstate    ::  ps,
                   files     ::  *Files,
                   iostate   ::  *IOState *(State ps)     }
```

Each interactive process has a fresh IOState environment. The IOState environment does not outlive the lifetime of an interactive process. With the IOState environment abstract event handlers can change the state of the Graphical User Interface elements at run-time. For this purpose the Event I/O library has an extensive set of functions.

Because an abstract event handler defines a transition of the process state (of type State ps), the type of an abstract event handler is :: *State ps $\rightarrow$ *State ps. So the algebraic type definitions of abstract devices are parameterized with *State ps.

(**c**) *Interactive processes* can be created and terminated with the predefined functions StartIO and QuitIO. In Clean the type definition of an n-ary function named f with arguments of type $\tau_1 \ldots \tau_n$ and result type τ is f :: $\tau_1\ \tau_2 \ldots \tau_n \rightarrow \tau$ (Common notation is f :: $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$).

```
StartIO      :: (IOSystem *(State ps)) ps *World -> *World
QuitIO       :: *(IOState *s) -> *IOState *s
```

StartIO is applied to a list of abstract device definitions that contain the definitions of the abstract event handlers of type :: *State ps $\rightarrow$ *State ps, the initial value of the program state of type ps, and the world environment. StartIO performs two actions: (1) creation of the proper environments for the interactive process, and (2) the evaluation of the interactive process until termination.

(1) Given the abstract device definitions StartIO creates the corresponding concrete Graphical User Interface elements. As a result the abstract devices appear to the user in their initial run-time state. The environment of type *IOState is created and filled with the run-time states of the concrete devices and their abstract event handlers.

(2) Then StartIO proceeds with the evaluation of the interactive process until termination. This is done by an *event loop*, which is a simple, recursive function. In each step the event loop retrieves a *concrete* event from the event stream environment, and if the concrete event should be interpreted as an abstract event, applies the corresponding abstract event handler to the *current* process state to obtain the *new current* process state. The *effect* of this transition is paired with the concrete event that triggered the transition. The event loop *terminates* as soon as the IOState component of the process state has been made *empty*. Abstract event handlers can make the IOState environment empty only with the QuitIO function. The result of StartIO is a new unique world environment that contains the new file system and event stream environments.

Figure 4 *a* summarizes the view a programmer has of the Event I/O system. Not included are the actual abstract device type definitions (TimerDef, MenuDef, WindowDef, and DialogDef), and the actual library functions defined on IOState with which the programmer changes the run-time state of the devices. Figure 4 *b* gives the internal definitions of StartIO, QuitIO, and IOState.

```
:: IOSystem s = [DeviceSystem s]

:: DeviceSystem s
     =   TimerSystem      [TimerDef       s]
     |   MenuSystem       [MenuDef        s]
     |   WindowSystem     [WindowDef      s]
     |   DialogSystem     [DialogDef      s]


:: State ps
     = {     pstate       :: ps,
             files        :: *Files,
             iostate      :: *IOState *(State ps)    }

:: IOState s


StartIO :: (IOSystem *(State ps)) ps *World -> *World
QuitIO :: *(IOState *s) -> *IOState *s
```

```
:: IOState s = {  devices      :: [DeviceState s],
                  events       :: *Events              }

StartIO :: (IOSystem *(State ps)) ps *World -> *World
StartIO ioSystem ps world
=     CloseWorld (s_n.files, s_n.iostate.events)
      where  s_n  = eventloop s_0
             s_0  = {pstate = ps, files = fs, iostate = io}
             io   = {devices = createDevices ioSystem, events = es}
             (fs, es) = OpenWorld world

finalState :: (State ps) -> Bool
finalState s = s.iostate.devices = [ ]

eventloop :: *(State ps) -> *State ps
eventloop s = loop finalState nextState s

loop :: (x -> Bool) (x -> x) x -> x
loop pred next x     | pred x  =    x
                               =    loop pred next (next x)

QuitIO :: *(IOState *s) -> *IOState *s
QuitIO io = { io & devices = [ ] }
```

4 a *The outside view of the Event I/O system.* ***4 b*** *The inside view of the Event I/O system.*

The arguments of a record can be *selected* by an extended form of *pattern-matching* and by the *field names* of the record. Consider for instance the IOState record type in figure 4 *b*. Suppose io is an expression of type IOState. On a pattern-match position the expression io={events=es} matches the variable es with the field events of io. On the right-hand-side of a function the expression io.devices selects the devices field of io, while io.events selects the events field of io. The arguments of a record are *updated* in the following way: the expression {io & events=es} is a record equal to io but the field events is replaced by the expression es.

We conclude this section with a small example that illustrates what an interactive Clean program looks like (figure 5). It is a very simple drawing program. The program state is a list of all drawn points. The program consists of two abstract devices: a menu device and a window device. There are three abstract event handlers: quit, track, and update. The abstract event handler quit, which is invoked when the menu item named Quit has been selected, simply terminates the program by applying QuitIO to its IOState component. The abstract event handler track, which is invoked for all mouse actions in its window, erases a point in the window and removes it from its program state (with the function delete) whenever the mouse is down and the option modifier key is pressed. It draws a point in the window and adds it to its program state whenever the mouse is down. The abstract event handler update, which is invoked whenever part of the window needs to be redrawn, redraws every drawn point.

```
:: DrawState = State [Point]

Start :: *World -> *World
Start world
=   StartIO ioSystem initialProgramState world
    where  ioSystem = [ MenuSystem       [PullDownMenu FileId "File" Able
                                              [MenuItem QuitId "Quit" (Key 'q') Able quit ] ],
                        WindowSystem     [FixedWindow 1 (0,0) "Picture" ((0,0),(200,100)) update
                                              [Mouse Able track, Cursor CrossCursor ] ] ]
           initialProgramState = [ ]

quit :: *DrawState -> *DrawState
quit s={ iostate = io } = {s & iostate = QuitIO io}

update :: UpdateArea *DrawState -> (*DrawState, [*Picture -> *Picture])
update _ s={ pstate=drawnPoints } = (s, map DrawPoint drawnPoints)

track :: MouseState *DrawState -> *DrawState
track (_, ButtonUp, _) s = s
track (point, _, OptionOnly) s={ pstate=drawnPoints, iostate=io }
=   { s & pstate = delete point drawnPoints, iostate = DrawInWindow 1 [ErasePoint point] io }
track (point, _, _) s={ pstate=drawnPoints, iostate=io }
=   { s & pstate = [point | drawnPoints], iostate = DrawInWindow 1 [DrawPoint point] io }
```

Figure 5 *A simple drawing program. The expression [x | xs] denotes a list with head item x and tail list xs.*

3 The Interleaved Event I/O system

In this section we describe how the Event I/O system can be equipped to handle interactive processes more flexible. The new system thus obtained is the *Interleaved Event I/O system*. New primitives are introduced into the system for *dynamic process creation* (section 3.1), and inter-process communication by *asynchronous message passing* (section 3.2), and *data sharing* (section 3.3). We demonstrate how these primitives can best be fit in with the abstract device and process state transition paradigms of the Event I/O system. In order to gain programming experience the Interleaved Event I/O system has been implemented. Each of the following subsections treats the consequences of these changes to the programming practice and the internal definitions of the I/O system.

3.1 Dynamic process creation

The first extension is the *dynamic creation of interactive processes*. Every interactive process can spawn interactive processes that will run *interleaved* with their father process. Every new interactive process runs independently of its father process, i.e. termination of the father process has no consequences for the child process, and vice versa.

The programmers view

The programmer creates interleaved interactive processes similarly to StartIO. The function NewIO :: (IOSystem *(State ps)) ps *(IOState *s)→*IOState *s creates the new interactive process and takes care that the new interactive process joins the evaluation of interactive processes. The type of NewIO is very similar to the type of StartIO. Environments of type IOState exist *inside* interactive processes only, so NewIO can be applied as part of an abstract event handler only. The contrary holds for StartIO: the world environment does not exist at evaluation of an interactive process, so it can be applied only *outside* interactive processes. The type of NewIO expresses that the process state type State ps of the child process is allowed to differ from the process state type s of the father process.

The system view

The implementation of the Clean I/O system changes in two major aspects. Firstly, the new I/O system needs to be able to handle an *arbitrary number of interactive processes*. Secondly, the *unique global* environments Files and Events need to be *shared* between all interactive processes. The changes are presented as follows: (**a**) the system regarded from the point of view of an interactive process, (**b**) the structure of the event loop, and (**c**) the definition of NewIO.

(**a**) Figure 6 gives the relevant types to an interactive process. In the interleaved system an interactive process is still a process state transition system on the process state type State. Whenever the interactive process is evaluated, its process state will consist of a program state, file system, and IOState component. The IOState component is also supplied with a list of the *other processes* of type Process. Because the *unique global* environments Files and Events are contained in the process state of this interactive process, the other processes cannot also contain these environments. Therefore Process ps is a polymorphic record type that contains only the *local* components of an interactive process. These local components are the *program state* of type ps and the *run-time states of the abstract devices*.

The type Process E.ps is an *existential type* of which the type variable ps is *existentially quantified*. The type [Process ps] enforces every element to have a program state of the *same type*, but different interactive processes employ program states of *different type*. A type definition can *hide* a type variable by prefixing the existential quantifier E. before the type variable. On the right-hand side of the type definition the type variable still enforces type equality, but the *scope* of the type variable is limited to the right-hand side of the type. The special purpose type constructor Void can be used as a type instance of existentially quantified type parameters. As a result, the type [Process Void] is a list of processes that have program states of different types. For an account of existential types in the Clean type system, see Plasmeijer and van Eekelen (1994), *in preparation*.

```
:: State ps      = {  pstate      :: ps,
                      files       :: *Files,
                      iostate     :: *IOState *(State ps)          }
:: IOState s     = {  devices     :: [DeviceState s],
                      events      :: *Events,
                      processes   :: [Process Void]                }
:: Process E.ps  = {  ppstate     :: ps,
                      pdevices    :: [DeviceState *(State ps)]     }
```

Figure 6 *The process administration types.*

(**b**) The interleaved event loop (figure 7) is more complicated than the simple event loop of section 2 because it must apply every new event to every process in the process list, and it must *(dis)connect* every *local* process component to the *global* environments. Therefore the interleaved event loop consists of two loops. The *outside loop* retrieves a new event from the event stream environment while there are processes to be evaluated. Given this event, the *inside loop* schedules these processes in round-robin order to compute their new process states. The *overall state* of the interleaved event loop is represented by the record type GlobalState that consists of the unique global environments Files and Events, and the current list of processes. The transformation of a process of type Process to a *process state* of type State is given by the function connect. The function disconnect is the *inverse* of connect (so disconnect•connect x = x). The process state transition function nextState given the event and the process state determines which, if any, of the available abstract event handlers of the process should be applied to the process state. The inside loop removes the process from the process list if the new process state is a terminal value. As a result, the interleaved event loop terminates as soon as all interactive processes have terminated.

The interleaved event loop is a *deterministically interleaved* state transition system. For each event retrieved from the event stream environment, the interactive processes are scheduled in round-robin order to compute the new value of their process state. The new value is given by one of its abstract event handlers. Which abstract event handler of the process should be applied to the process state depends on the actual value of the event. So the interleaving of interactive processes is deterministic.

```
:: GlobalState = {   gFiles          :: *Files,
                     gEvents         :: *Events,
                     gprocesses      :: [Process Void]     }

StartIO :: (IOSystem *(State ps)) ps *World -> *World
StartIO ioSystem ps world
=    CloseWorld (gsn.gFiles, gsn.gEvents)
     where  gsn          = eventloop gs0
            gs0          = { gFiles = fs, gEvents = es, gprocesses = [process] }
            process      = { ppstate = ps, pdevices = createDevices ioSystem }
            (fs, es)     = OpenWorld world

eventloop :: *GlobalState -> *GlobalState
eventloop gs
=    loop emptyGState nextGState gs
     where  emptyGState gs = gs.gprocesses = [ ]

            nextGState gs={ gEvents=es }
            =    gs1
            where  (e, es1) = nextevent es
                   (_, gs1) = loop procsdone (nextproc e) (1, { gs & gEvents=es1 })

                   procsdone (i, gs) = i > length gs.gprocesses

                   nextproc e (i, { gFiles=fs, gEvents=es, gprocesses=procs })
                   =    (length procs2-length procs+i+1, { gFiles=fs1, gEvents=es1, gprocesses=procs2 })
                   where  procs2                   = if (proc.devices=[ ]) procs1 [proc | procs1]
                          ((proc, procs1),fs1,es1) = disconnect•nextState e•connect (remove i procs, fs, es)
```

Figure 7 *The definition of the interleaved event loop.*

(**c**) In this set-up the definition of NewIO can be straightforward: it maps the abstract device definitions of the interactive process to concrete devices, creates the process record value, and inserts it into the process list that is contained in its IOState argument, so that next time the event loop will schedule the new interactive process for evaluation.

```
NewIO :: (IOSystem *(State ps)) ps *(IOState *s) -> *IOState *s
NewIO ioSystem ps io={ processes=procs } = { io & processes = [ { ppstate = ps, pdevices = createDevices ioSystem } | procs ]}
```

3.2 Asynchronous message passing

In this step we add a *polymorphic, type-safe, asynchronous message passing mechanism* to the Interleaved Event I/O system. Interactive processes can send messages to any other interactive process, provided they have the *identification* of that process. For this purpose the I/O system generates a unique identification for every interactive process. The content of a message can be any typeable expression. The type system is applied to enforce type-safe message passing: it is impossible for a correctly typed interactive process to send messages of the wrong type. In the I/O system, messages are considered to be *abstract events*. Conform the Event I/O paradigm of abstract event handling by abstract devices, message events are dealt with by a new abstract device, the *receiver device*.

The programmers view

Figure 8 presents the changes to the Interleaved Event I/O system that concern the programmer. There are three important changes: (**a**) the abstract device type definitions have been extended with the abstract receiver device, (**b**) at creation of a receiver a parameterized identification is returned, and (**c**) the library functions have been extended with a type-safe message passing function.

```
:: ReceiverDef       m s = Receiver SelectState (ReceiverFunction m s)
:: ReceiverFunction m s = m s -> s
:: IOId m

OpenReceiver      :: (ReceiverDef m *s)    *(IOState *s) -> (IOId m, *IOState *s)
CloseReceiver     :: (IOId m)              *(IOState *s) -> *IOState *s
```

```
Send                :: (IOId m) m               *(IOState *s) -> *IOState *s
```

Figure 8 *The predefined abstract receiver device type definition and the new functions for the programmer.*

(**a**) The algebraic type ReceiverDef that defines the abstract receiver device is very straightforward. A receiver can be Able or Unable (the SelectState attribute). The *abstract event handler* that should be evaluated in case a message event arrives for the receiver is the *receiver function*. The receiver function accepts messages of a given *polymorphic type* m.

(**b**) Interactive processes can open and close an arbitrary number of receivers dynamically with the functions OpenReceiver and CloseReceiver. In order to identify the addressee of a message uniquely, OpenReceiver generates a *unique identification* for each receiver. The identification of a receiver that accepts messages of type m is a value of type *IOId m*. IOId is an *abstract data type* that is *parameterized* with the message type of the receiver.

(**c**) Interactive processes send a message with the function Send which requires the identification IOId m of the receiver and a message of the corresponding type m. Send inserts the pair of receiver identification and message in the event stream environment contained in the IOState argument.

The system view

The major change to the Interleaved Event I/O system is the definition of the receiver device. The receiver device of an interactive process checks, when applied to a message event and the current process state, whether the identification in the message event corresponds to one of its (*able*) receivers. If this is the case then the response of the interactive process is defined by the application of the receiver function f to the actual message m and the current process state s, so the response is f m s.

This message passing mechanism is type-safe. In order to send a message to a receiver that accepts messages of type m, the corresponding identification is required. Receiver identifications are created with OpenReceiver only and uniquely identify the receiver. The identification is parameterized with the message type m of that receiver. So the message type m of a receiver identification equals the message type m of the receiver function identified by the identification. The type system enforces type equality between the message type of the process identification and the message being sent.

3.3 Data sharing

The idea of sharing the file system and event stream environments between interactive processes can be generalized to *sharing an arbitrary data structure between a group of interactive processes*. Analogous to inter-process communication with file I/O on the file system environment, and message passing on the event stream environment, interactive processes that share a data structure can *communicate by writing and reading the shared data structure*. For this reason this kind of inter-process communication is called *data sharing*. Access to the shared data structure is *atomic*.

The programmers view

Every interactive process defines a public component of arbitrary type that can be used lateron as the shared data structure (figure 9). So the process state of every interactive process consists of a *local* component and a *public* component. The types of the functions StartIO and NewIO change in a non-essential way. An interactive process can spawn a so called *shared* interactive process with the new function ShareIO. Analogous to NewIO, the shared interactive process runs *interleaved* with all other interactive processes. The difference is that the types of the public components of the process states of both interactive processes must be equal. The public component will be shared during evaluation of both processes. Every new shared interactive process that is spawned by any interactive process will also share the same public component. The interactive processes that share the same public component form a *group of interactive processes*. It should be observed that ShareIO does not define some initial value of type p for the public component because this value already exists.

```
:: State l p = {    localstate   :: l,
                    publicstate  :: p,
                    files        :: *Files,
                    iostate      :: *IOState *(State l p)    }

StartIO     :: (IOSystem *(State l p)) (l, p)
               *World -> *World
NewIO       :: (IOSystem *(State l p)) (l, p)
               *(IOState s) -> *IOState s
ShareIO     :: (IOSystem *(State l p)) l
               *(IOState *(State l' p)) -> *IOState *(State l' p)
```

Figure 9 *The outside view.*

```
:: IOState s          = {  devices      :: [DeviceState s],
                           events       :: *Events,
                           myGroup      :: [Process Void Void],
                           otherGroups:: [Group Void Void]          }
:: GlobalState        = {  gFiles       :: *Files,
                           gEvents      :: *Events,
                           ggroups      :: [Group Void Void]        }
:: Group E.l E.p      = {  public       :: p,
                           processes    :: [Process l p]            }
:: Process E.l E.p    = {  pstate       :: l,
                           pdevices     :: [DeviceState *(State l p)]  }
```

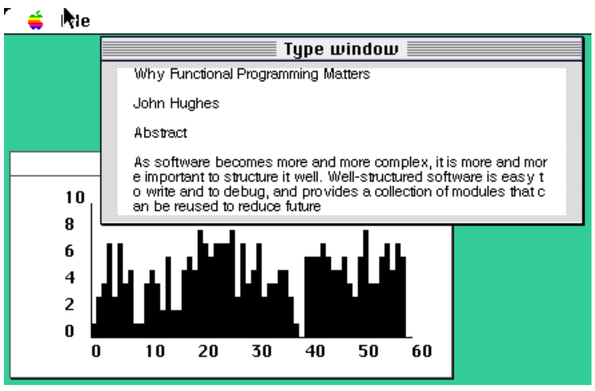
Figure 10 *The inside view.*

The system view

The extension of the Interleaved Event I/O system with data sharing complicates only the implementation of the interleaved event loop. The basic difference between the data sharing interleaved event loop and the interleaved event loop as defined in section 3.1 is that the new event loop traverses a *list of lists of processes* rather than one *list of processes*. We omit the definition of the event loop for reasons of space and give the internal data types only (figure 10).

4 Example

In this section we present an example of how an interactive Graphical User Interface application can be constructed with the primitives that have been discussed in the previous sections. The program monitors the typing speed of a user for a period of one minute. During the typing session it shows the key hit rate per second. Figure 11 gives a snapshot of the application running on a Macintosh system. The program consists of two interactive processes: the *typist process* and the *monitor process* that communicate by asynchronous message passing. Some minor functions and constants have been omitted in the program code for reasons of brevity.



Why Functional Programming Matters

John Hughes

Abstract

As software becomes more and more complex, it is more and mor
e important to structure it well. Well-structured software is easy t
o write and to debug, and provides a collection of modules that c
an be reused to reduce future

Figure 11 *A snapshot of the typing monitor running.*

The *monitor process* (figure 12 *a*) draws during a session in its window the progress of the user. The monitor process shares no data, so its *public* process state type is Empty (:: Empty = **Nil**). The monitor process is the initial interactive process created by the program. Its first action is to create the typist process. The *local* process state is the record type Local. The field count holds the number of key hits per second, counts holds the list of counts so far, and time holds the elapsed time of the session. The monitor process is controlled by the messages of type MonitorMessage it receives from the typist process. If it receives the **StartSession** message, it sets the local process state to its initial value, and *enables* the timer. It *disables* the timer if it receives the **EndSession** message. For each **KeyHit** message it increments count. Finally, on acceptence of the **QuitMonitor** message it quits the monitor process. The timer event handler drawKeyHits, when enabled, is evaluated once every second. It appends count to counts, sets count to zero, increments time, and draws the count value in the window.

```
:: MonitorMessage        = StartSession | EndSession | KeyHit | QuitMonitor
:: Local                  = {  count         :: Int,
                               counts        :: [Int],
                               time          :: Int      }
:: MonitorState           = State Local Empty

Start :: *World -> *World
Start world
=    StartIO ioSystem (initialLocal, Nil) initialIO world
     where  ioSystem     = [  WindowSystem [FixedWindow MonitorWindowId (0,0) "Monitor" MonitorDomain
                                                   updatemonitor [WindowNoGoAway]],
                              TimerSystem [Timer DrawHitsId  Unable  Second drawKeyHits],
            initialIO    = [openChannel]

initialLocal = { count=0, counts=[ ], time=0 }

openChannel :: *MonitorState -> *MonitorState
openChannel monitor={ iostate=io } =    openTypist talkTo { monitor & iostate=io1 }
                                        where    (talkTo, io1) = OpenReceiver (Receiver Able  receive) io

receive :: MonitorMessage *MonitorState -> *MonitorState
receive StartSession monitor={ iostate=io }
=    DrawInWindowFrame MonitorWindowId updatemonitor monitor1
     where  monitor1 = { monitor & localstate=initialLocal, iostate=EnableTimer DrawHitsId io }
receive KeyHit            monitor={ count=c    }    = { monitor & count=c+1 }
receive EndSession        monitor={ iostate=io }    = { monitor & iostate=DisableTimer DrawHitsId io }
receive QuitMonitor       monitor={ iostate=io }    = { monitor & iostate=QuitIO io }

drawKeyHits :: TimerState *MonitorState -> *MonitorState
drawKeyHits _ monitor={ localstate={ count=c, counts=cs, time=t }, iostate=io }
=    {monitor & localstate={ count=0, counts=append c cs, time=t+1 }, iostate=DrawInWindow 1 [drawCount t c] io }
```

Figure 12 a *The main code of the monitor process.*

The *typist process* (figure 12 *b*) controls the user input (selection of the commands Run and Quit, and keyboard handling by the, initially disabled, abstract event handler typeAndSendKeys). The initially disabled timer is used as a one-minute stopwatch. When created by the monitor process the typist process stores the IOId of the monitor process in the *local* process state component of type Local. Local also contains the text lines the user has typed during a session. A session is started with *Run*. Run sets the local process state to its initial value, *enables* keyboard handling of the window and *enables* the timer, *disables* menu selection of itself, and *sends* the **StartSession** message to the monitor process. A session terminates after one minute triggered by evaluation of the *timer function* endOfSession which *disables* the keyboard handling of the window and *disables* the timer, *enables* selection of the Run command, and *sends* the **EndSession** message to the monitor process. During a session, typeAndSendKeys sends a **KeyHit** message to the monitor process for every key that has been pressed. Finally, the Quit command sends the **QuitMonitor** message to the monitor process and terminates the typist process.

```
:: Local         = {  lines      :: [String],
                      talkTo     :: IOId MonitorMessage       }
:: TypeState     = State Local Empty

openTypist :: (IOId MonitorMessage) *(State l p) -> *(State l p)
openTypist ioId state={ iostate=io }
=    { state & iostate=NewIO ioSystem (initialLocal,  Nil) initialIO io }
     where  ioSystem = [MenuSystem  [PullDownMenu FileId "File"  Able [
                                         MenuItem RunId "Run" (Key 'r') Able run,
                                         MenuSeparator,
                                         MenuItem QuitId "Quit" (Key 'q')  Able quit]],
                        WindowSystem [FixedWindow TypeWindowId (0,0) "Type window" TypeDomain updatewindow
                                         [WindowKeyboard Able  typeAndSendKeys,  WindowNoGoAway]],
                        TimerSystem [Timer StopwatchId  Unable  OneMinute endOfSession]]
            initialLocal   = { lines=[""], talkTo=ioId }
            initialIO      = []

run :: *TypeState -> *TypeState
run state={ localstate={ talkTo=monitor }, iostate=io }
=    DrawInWindowFrame 1  updatewindow { state & { localstate & lines=[""] }, iostate=io1 }
     where  io1 = ChangeIOState [  EnableKeyboard TypeWindowId,
                                   EnableTimer StopwatchId,
                                   DisableMenuItems [RunId],
                                             Send monitor StartSession] io

quit :: *TypeState -> *TypeState
quit state={ localstate={ talkTo=monitor }, iostate=io } = { state & iostate=QuitIO (Send monitor QuitMonitor io) }

endOfSession :: TimerState *TypeState -> *TypeState
endOfSession _ state={ localstate={ talkTo=monitor }, iostate=io }
=    { state & iostate=ChangeIOState [  DisableKeyboard TypeWindowId,
                                        DisableTimer StopwatchId,
                                        EnableMenuItems [RunId],
                                        Send monitor EndSession] io }

typeAndSendKeys :: KeyboardState *TypeState -> *TypeState
typeAndSendKeys (key, KeyDown, _) state={ localstate=local, iostate=io }
where local={ lines=text, talkTo=monitor }
=    { state & localstate={ local & lines=text1 }, iostate=Send monitor KeyHit (DrawInWindow TypeWindowId drawText io) }
     where   (text1, drawText) = addCharToText key text
typeAndSendKeys _ state = state
```

Figure 12 b *The main code of the typist process.*

Finally, both processes can be restructured easily to communicate by data sharing rather than message passing. Let count be the shared data structure. The *typist process* increments the shared count for every key hit instead of sending the **KeyHit** message. The *monitor process* reads and sets the shared count, and removes count from its local process state. The **KeyHit** message can also be removed.

5 Related work

The research described in this paper stems from research on functional I/O systems, and more specifically the incorporation of Graphical User I/O into a functional language. Early work in this area is the work on *dialogue combinators* by Dwelly (1989). Recent work is the *FUDGETS* system by Carlsson and Hallgren (1993). The issue of dynamic process creation has not been considered in the dialogue combinator system, and the FUDGETS system dynamic process creation is very limited. Noble and Runciman (1994) give a comparison between the FUDGETS system and the Clean Event I/O system.

The area of functional operating systems offers more closely related work with respect to dynamic process creation and inter-process communication. Closely related work in this area is the Kent Applicative Operating System (KAOS) project by Turner (1990) which is a framework for a functional operating system. The system is based on earlier work by Stoye (1984). Both systems allow dynamic creation of functional processes. The inter-process communication is based on the *sorting office* concept introduced by Stoye. Essentially, the sorting office implements a *non-deterministic merge* of all messages *outside* the language. In the Clean (Interleaved) Event I/O scheme the event stream environment is the infinite stream of all input-events ever in which message events can be inserted by interactive processes in a functional way. No additional non-functional merge is required.

6 Conclusions

In this paper we have investigated how a single state transition system (the Event I/O system) can be turned into an *interleaved* state transition system (the Interleaved Event I/O system) that is *dynamically* composed of single state transition definitions (interactive processes). We have added two forms of inter-process communication in this system, namely *asynchronous message passing* and *data sharing*. Both forms of communication are type-safe and polymorphic. The construction is done entirely on the functional level. The Interleaved Event I/O system can be regarded as an *operational semantics* of dynamic functional processes. The inter-process communication gains by the fact that the system has been defined in a functional framework. Both forms of communication can use higher-order functions and arbitrarily complex data types straightforwardly.

We have gained some programming experience with the Interleaved Event I/O system. Some of the programs we have written are simulations of parallel systems: a talk application, and some distributed games. Another class of programs that can be suitably dealt with in this framework are process control applications: we have written a program that controls the temperature of a water tank.

7 Current and future work

Current work involves aspects of the interleaved state transition system, such as the inclusion of existing (non-functional) applications in the framework, and how other forms of message passing such as synchronous message passing, remote invocation, and remote procedure calling, can be defined in the system. The suitability of the primitives still requires further investigation. The main topic of our future work will be to use the Interleaved Event I/O system as a basis for a *distributed* Event I/O system.

Acknowledgements

The authors would like to thank Marko van Eekelen for the many discussions during this research, and the useful comments on the content of the paper.

References

Achten, P.M., van Groningen J.H.G., and Plasmeijer, M.J. 1993. 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 June 1992. Workshop Notes in Computer Science. Springer-Verlag,Berlin,1993, pp 1-17.

Achten, P.M. and Plasmeijer, M.J. 1994. The Ins and Outs of Clean I/O. To appear in the *Journal of Functional Programming*.

Barendregt, H.P., Eekelen van, M.C.J.D., Glauwert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., and Sleep, M.R. 1987. Term Graph Rewriting. In Bakker, J.W. de, Nijman, A.J., and Treleaven, P.C. eds. Proceedings of Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, LNCS 259, Vol.II. Springer-Verlag, Berlin, pp. 141-158.

Barendsen, E. and Smetsers, J.E.W. 1993. Conventional and Uniqueness Typing in Graph Rewrite Systems. In Shyamasundar, R.K. ed. *Proceedings of the Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, 15–17 December 1993, Bombay, India. LNCS **761**. Springer-Verlag, Berlin, pp. 41-51.

Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, Plasmeijer, M.J., and Barendregt, H.P. 1987. 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.

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

Dwelly, A. 1989. Functions and Dynamic User Interfaces. In *Proceedings of Fourth International Conference on Functional Programming Languages and Computer Architectures*, Imperial College, London, September 11-13, 1989, pp. 371-381.

Groningen, J.H.G. van, Nöcker, E.G.J.M.H., and Smetsers, J.E.W. 1991. Efficient Heap Management in the Concrete ABC Machine. In Glaser, Hartel eds, *Proceedings of Third International Workshop on Implementation of Functional Languages on Parallel Architectures*. University of Southampton, UK. Technical Report Series CSTR 91-07.

Henderson, P. 1982. Purely Functional Operating Systems. In Darlington, J., Henderson, P., Turner, D.A. eds., *Functional programming and its applications*, Cambridge University Press, pp. 177-192.

Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. 1991. 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.

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

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

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

Plasmeijer, M.J. and van Eekelen, M.C.J.D. 1994. Clean 1.0 Reference Manual. *Technical Report*, in preparation. University of Nijmegen, The Netherlands.

Smetsers, J.E.W., Nöcker, E.G.M.H, Groningen, J.H.G. van, and Plasmeijer, M.J. 1991. Generating Efficient Code for Lazy Functional Languages. In Hughes, J. ed, *Proceedings of Fifth International Conference on Functional Programming Languages and Computer Architecture* Cambridge, MA, USA, LNCS **523**, Springer-Verlag pp. 592-617.

Smetsers, J.E.W., Barendsen, E., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. 1993. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In *Proceedings Workshop Graph Transformations in Computer Science*, Schloss Dagstuhl, January 4-8, 1993. Lecture Notes in Computer Science, Springer-Verlag, Berlin.

Stoye, W.R. 1984. A new scheme for writing functional operating systems. *Technical Report 56*, Computer Laboratory, Cambridge University, 1984.

Thompson, S. 1990. Interactive Functional Programs. A Method and a Formal Semantics. In Turner, D.A. ed., *Research topics in Functional Programming*, Addison-Wesley Publishing Company, pp. 249-285.

Turner, D.A. 1990. An Approach to Functional Operating Systems. In Turner, D.A. ed., *Research topics in Functional Programming*, Addison-Wesley Publishing Company, pp. 199-217.