3

Interleaved Interactive Functional Processes

In this chapter 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: *data sharing* and *message passing*. In our system both forms of communication are polymorphic and type-safe. Both forms of communication can use 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 can be regarded as an executable *operational semantics* of a mini operating system for dynamic, interactive, functional processes. The system has been implemented and will become part of the new release of Clean.

3.1 Introduction

Research in the area of functional programming languages is experiencing an increasing attention to the incorporation of I/O into the functional programming paradigm (Henderson, 1982; Dwelley, 1989; Turner, 1990; Thompson, 1990; Peyton Jones and Wadler, 1993; Carlsson and Hallgren, 1993). In this chapter 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 the start, the major part of the research has focused on basic issues such as how at all to incorporate I/O 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 (1995*a*) – see Chapter 2.

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 to improve on the modular structure of the program.

The system we introduce in this chapter, the *interleaved event I/O system*, is used to study how to construct programs that consist of many processes at a time, and what suitable inter-process communication methods are. These forms of communication are *data sharing* and *synchronous* as well as *asynchronous message passing*. Both forms are type-safe and polymorphic. It should be noted that we do not study concurrent processes in this chapter: the processes will be evaluated in a deterministically and interleaved way.

The semantics of the interleaved event I/O system in this chapter is defined in Clean. The reasons to use Clean as a specification language is that it has a well-defined syntax and semantics (Clean is based on term graph rewriting), it allows specifications to be type-checked, compiled, and tested (as such specifications are readily executable) and last but not least, by construction, the specified system is purely functional such that the advantages of functional programming remain. In particular, interactive processes can communicate higher-order functions and arbitrary data structures to other processes. The specification of the interleaved event I/O system can be used as a framework for the actual implementation.

This chapter presents the interleaved event I/O system in a number of steps. Section 3.2 starts with an introduction to the basic event I/O system in terms of the interleaved framework. Section 3.3 explains how to handle many processes at a time. Section 3.4 introduces inter-process communication by data sharing, and Section 3.5 introduces inter-process communication by message passing. The Sections 3.2 upto 3.5 are divided in two subsections. In each case the first subsection presents the topic from the programmer's point of view, and the second from the system's point of view. Section 3.6 presents an example. Section 3.7 presents related work. Conclusions are drawn in Section 3.8, and Section 3.9 concludes with current and future work.

3.2 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 chapter are written in Clean 1.0 (Plasmeijer and van Eekelen, 1995). Most of the language constructs used in Clean 1.0 are customary in other functional languages such as Miranda and Haskell. Where appropriate, the text includes remarks on particular aspects 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, files, menus, windows, or timers). The symbol '`*`' is a *type attribute* indicating 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 3.1). 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 with the function `openworld`::`*World` → (`*Files`, `*Events`). These environments can create a new unique world environment with the function of reverse type `closeworld`::(`*Files`, `*Events`) → `*World`. The file system environment contains the individual file environments (of type `File`) for file I/O. Files can be opened for writing/reading (in which case they obtain the uniqueness attribute and are of type `*File`) or for read-only. The event stream environment is discussed in the next section.

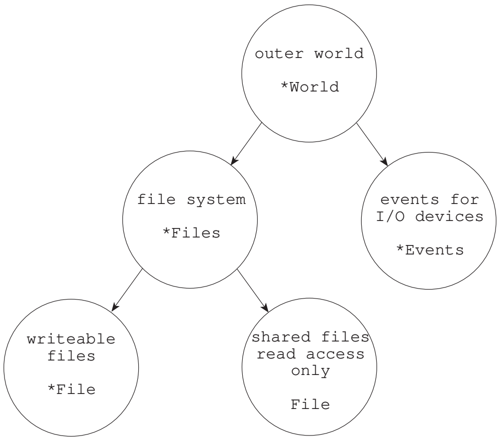


Figure 3.1 The Clean environment hierarchy.

3.2.1 The programmer's view

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**) initialize the event I/O system by applying the predefined function `OpenIO` to these definitions. `OpenIO` 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 three abstract devices: the window (including dialogues), menu, and timer device. Abstract interface elements are specified by expressions that yield a value of predefined algebraic type (Figure 3.2). These algebraic types specify for each kind of abstract device how the individual elements of that abstract device should be defined. The *abstract event handlers* that are contained in the abstract device definitions are *transition functions* of type `ps` → `ps` of some *process state* of type `ps` (see later on). Therefore the type definitions of abstract devices are parameterized with the type variable `ps`. Note that in Clean the type definition of an n-ary function with name *f* and arguments of type $\tau_1 \ldots \tau_n$ and result type τ is $f :: \tau_1 \ldots \tau_n \rightarrow \tau$ rather than the more common notation $f :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$. As an illustration of an abstract device definition, Figure 3.3 gives a typical example of a menu definition. The picture below the definition shows the concrete device in the case of the menu definition being mapped to a Macintosh system.

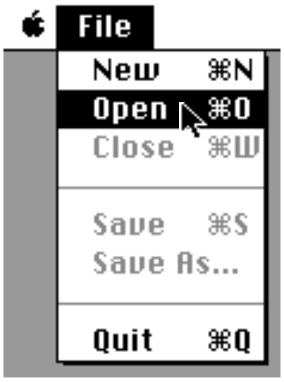
```
::  MenuDef ps
=   Menu Title           [MenuElement ps] [MenuAttribute ps]
::  MenuElement  ps
=   SubMenuItem  Title [MenuElement ps] [MenuAttribute ps]
|   MenuItem     Title                  [MenuAttribute ps]
|   MenuSeparator

::  MenuAttribute ps                      // Default:
=   MenuId            Id                  // no Id
|   MenuSelectState   SelectState         // menu(item) Able
|   MenuShortKey      KeyCode             // no KeyCode
|   MenuAltKey        Index               // no AltKey
|   MenuMarkState     MarkState           // NoMark
|   MenuFunction      (IOFunction     ps) // no shortcut
|   MenuModsFunction  (ModsIOFunction ps) // no shortcut
::  IOFunction     ps :==           ps -> ps
::  ModsIOFunction ps :== Modifiers ps -> ps
```

Figure 3.2 Clean's predefined algebraic type `MenuDef` to define individual menus. Type variables start in lowercase. Type `[a]` means a list of a. Symbols printed in **boldface** are alternative data constructors of the algebraic type. Comments start with `//`.

```
Menu "File" [
   MenuItem "New"       [MenuFunction new,  MenuShortKey 'n'],
   MenuItem "Open…"     [MenuFunction open, MenuShortKey 'o'],
```

```
MenuItem "Close"    [MenuFunction close,MenuShortKey 'w',
                     MenuSelectState    Unable],
MenuSeparator,
MenuItem "Save"     [MenuFunction save, MenuShortKey 's',
                     MenuSelectState    Unable],
MenuItem "Save As…" [MenuFunction saveAs,
                     MenuSelectState    Unable],
MenuSeparator,
MenuItem "Quit"     [MenuFunction quit, MenuShortKey 'q']] []
```

Figure 3.3 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 3.3. 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 the predefined parameterized *record type* `PState l` (Figure 3.4) and consists of three components. The first component is the *program state* of arbitrary type `l` which reflects the logical state of the interactive process. For notational conciseness the type variables `l` and `m` are used for the local type variable in (function) type definitions. The second component is the file system environment. Finally, the most important component is the `IOState` environment. This environment provides the interactive process with abstract access to the graphical user interface elements. Each interactive process has a private `IOState` environment which does not outlive the lifetime of the interactive process.

```
:: *PState  l =  { pLocal   :: l,
                   pFiles   :: *Files,
                   pIOstate :: *IOState l}
:: *IOState l
```

Figure 3.4 The process state type and the `IOState` environment. The type `IOState` is an abstract data type, which is indicated in Clean by giving only the left-hand-side type definition.

Because abstract event handlers define transitions of the process state of type (`PState l`) their types are `*(PState l)` → `*(PState l)`. So the algebraic type definitions of abstract devices are parameterized with `*(PState l)`. Given 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 to manipulate windows, menus and timers. Interface elements are identified by special identification values of abstract type `Id`. These values are generated by the library function `IOGetIds` that yields a stream of fresh identification values.

Finally, a note on record types in Clean. A record type is an algebraic type where selection is done using *field names* instead of using position matching. For instance, let `ps` be an expression of type `PState`. The expression `ps=:{pFiles = fs}` matches the variable `fs` with the field `pFiles` of `ps`. On the right-hand-side of a function, the expression `ps.pFiles` selects the `pFiles` field of `ps`. The arguments of a record are updated as follows: the expression `{ps & pFiles=fs}` denotes a record equal to `ps` but the field `pFiles` has value `fs`.

(**c**) Interactive processes are programmed as state transition systems. The state is the process state of type `PState l`, and the state transitions are the abstract event handlers of type `*(PState l)` → `*(PState l)`. The evaluation of this state transition system is done by the library function `OpenIO` (Figure 3.5). `OpenIO` is applied to an initial set of actions, an initial value of the local program state, and the `World` environment. It evaluates the interactive process until termination and then yields a new `World` value. An interactive process requests termination by applying the library function `QuitIO` to its `IOState` environment.

```
:: InitIO ps :== [ps -> ps]
```
The initial actions

```
OpenIO:: (InitIO *(PState .l)) .l *World -> *World
```
Start the first interactive process

```
QuitIO :: *(IOState .l) -> *(IOState .l)
```
Terminate evaluation of this process

Figure 3.5 The programmer's view of the event I/O system. To distinguish synonym type definitions from algebraic type definitions with one alternative constructor, synonym type definitions use the `:==` symbol.

3.2.2 The system view

As mentioned in Section 3.2.1, low-level event handling is done by the event I/O system, not by the programmer. The programmer provides the system with abstract device definitions, containing abstract event handlers, and requests the evaluation of the interactive process. The system in its turn does the following:

(**a**) map the abstract device definitions to the concrete devices,

(**b**) store the concrete devices, and

(**c**) evaluate the thus defined interactive process by handling concrete events and apply the corresponding abstract event handlers.

(**a**) The abstract device definitions provide a platform independent definition of the elements a programmer can use to construct interactive programs. Eventually these definitions must be mapped to concrete graphical user interface elements. The data type of the concrete elements does not only contain platform dependent graphical user interface information, but also the abstract event handlers. `DeviceState ps` is the result type of mapping abstract devices to concrete devices. It is parameterized with the process state type `ps` because it holds the abstract event handlers of type `ps` → `ps`.

(**b**) Figure 3.6 presents the internal definition of the `IOState` environment. The main components of the `IOState` environment are the run-time states of the abstract devices and the event stream environment. The `Id` field is used to generate fresh `Id` values, as mentioned in Section 3.2.1(**b**) with the library function `IOGetIds`. Every `Id` value can be used as a seed to generate new `Id` values with the function `GenIds` :: `*Id` → ([`*Id`], `*Id`). An initial `Id` value is obtained by the function `InitId` :: `*Id` (both functions are not in scope of the programmer). The implementation of `IOGetIds` is straightforward:

```
IOGetIds :: *(IOState .l) -> ([*Id], *IOState .l)
IOGetIds io
=      (ids, {io & ioId=id})
where (ids, id) = GenIds io.ioId
```

Finally, the initial actions of the interactive process are stored so they can be evaluated before the first abstract event handler of the interactive process is evaluated.

```
:: *IOState l = { ioDevices:: [DeviceState *(PState l)],
                  ioInitIO :: InitIO       *(PState l),
                  ioId     :: *Id,
                  ioEvents :: *Events                   }
```

Figure 3.6 The `IOState` environment.

The `PState` record as used by the abstract event handlers actually contains the complete state of a program. However, it reflects this state as seen by the programmer. Internally, this state information is restructured to make event handling and process administration easier. Internally, the run-time state is represented by the record type (`GState l`) (see Figure 3.7) that contains the global environments `Files` and `Events` and the local process administration of type (`Process l`). The local process components are the program state, the initial actions, the run-time states of the abstract devices, and the identification value.

There is a straightforward correspondence between `GState` and `PState` data types. The transformation from `GState` to `PState` and vice versa is given by the functions `connect` and `disconnect`. These mappings become more involved in the following sections.

```
:: *GState l  = { gFiles   :: *Files,
                  gEvents  :: *Events,
                  gProcess :: *Process l                 }
:: *Process l = { ppLocal  :: l,
                  pId      :: *Id,
                  pInitIO  :: InitIO       *(PState l),
                  pDevices :: [DeviceState *(PState l)] }

connect :: *(GState .l) -> *(PState .l)
connect {gFiles,gEvents,gProcess={ppLocal,pInitIO,pDevices,pId}}
```

```
=   { pLocal   = ppLocal,
      pFiles   = gFiles,
      pIOState = { ioDevices = pDevices,
                   ioInitIO  = pInitIO,
                   ioId      = pId,
                   ioEvents  = gEvents }}

disconnect :: *(PState .l) -> *(GState .l)
disconnect {pLocal,pFiles,pIOState={ioDevices,ioInitIO,ioId,ioEvents}}
=   { gFiles   = pFiles,
      gEvents  = ioEvents,
      gProcess = { ppLocal  = pLocal,
                   pInitIO  = ioInitIO,
                   pDevices = ioDevices,
                   pId      = ioId     }}
```

Figure 3.7 The internal process administration types.

(**c**) Figure 3.8 presents the complete definition of `OpenIO`. `OpenIO` first creates the proper `GState` record and then evaluates the interactive process until termination.

```
OpenIO :: (InitIO *(PState .l)) .l *World -> *World
OpenIO initIO local world
=     closeworld (gsn.gFiles,gsn.gEvents)
where (fs,es) = openworld world
      proc    = { ppLocal  = local,
                  pId      = InitId,
                  pInitIO  = initIO,
                  pDevices = []   }
      gs0     = { gFiles   = fs,
                  gEvents  = es,
                  gProcess = proc }
      gsn     = loop emptyGS nextGS gs0

      emptyGS gs=:{gProcess={pDevices=[]}} = (True, gs)
      emptyGS gs                           = (False,gs)

      nextGS  gs
      =     gs2
      where (e, es1) = nextEvent gs.gEvents
            gs1      = {gs & gsEvents=es1}
            gs2      = dispatch e gs1

            dispatch e gs
            =     (disconnect o nextPState e o doInitIO o connect) gs

            doInitIO ps=:{pIOState={ioInitIO}}
            =     ps2
            where ps1 = seq ioInitIO ps
                  ps2 = {ps1 & pIOState={ps1.pIOState & ioInitIO=[]}}

loop :: (.x -> (Bool,.x)) (.x -> .x) .x -> .x
loop final next x
|   isfinal   = x1
|   otherwise = loop final next (next x1)
where
    (isfinal, x1) = final x
```

```
QuitIO :: *(IOState .l) -> *(IOState .l)
QuitIO io = {io & ioDevices=[]}
```

Figure 3.8 The system view of the event I/O system (operator o is function composition).

The evaluation of the interactive process is done by an *event loop*, which is a simple recursive function defined by `loop`. In each step the event loop retrieves a concrete event from the event stream environment with `nextEvent`. It then transforms the `GState` record into the `PState` record. The initial actions are evaluated only the first time by `doInitIO`. Then `nextPState` determines whether the concrete event should be interpreted as an abstract event, and applies the corresponding abstract event handler to the current process state to obtain the new process state. Obviously `nextPState` depends on the concrete kinds of events and device implementations. The effect of this transition is paired with the concrete event that triggered the transition. Finally, the new `PState` record is transformed into a `GState` record.

The event loop *terminates* as soon as the process state contains no more devices, and therefore no more abstract event handlers. This state can be reached only if one of the abstract event handlers has made the `IOState` environment *empty* with the `QuitIO` function *or* by simply not opening any device in the initial actions of the process (in which case the event loop terminates immediately). The result of `OpenIO` is a new unique world environment that contains the new file system and event stream environments.

We conclude this section with a small example that illustrates what an interactive Clean program looks like (Figure 3.9). The program monitors the keyhit-rate of a user for a period of one minute. The initially disabled timer is used as a one-minute stopwatch. The local program state of type `Local` contains the text lines the user types during a session and the character count. The initial action of the program is to create the window, menu and timer. A session is started with the `Run` command, which sets the program state to its initial value, enables keyboard handling of the window and enables the timer, and disables menu selection of itself. A session terminates after one minute triggered by evaluation of the timer function `end` which disables the keyboard handling of the window and disables the timer, enables selection of `Run`, and prints the character count. During a session, `handleKeys` prints every key in the window and increments the current count. Finally, `Quit` terminates the process.

```
::  Local = { lines :: [String],
              count :: Int
            }

Start :: *World -> *World
Start world = OpenIO [initIO] initLocal world

initLocal :: Local
initLocal = {lines=[],count=0}

initIO :: *(PState Local) -> *(PState Local)
initIO ps = seqPIO [OpenTimer  timer,
                    OpenWindow window,
                    OpenMenu 0 menu] ps
where
    ([wId,mId,tId:_],io) = IOGetIds ps.pIOState
    ids    = (wId,mId,tId)
    window = Window "Type window" TypeDomain []
                [WindowKeys   Able (handleKeys wId),
                 WindowId     wId,
```

```
                  WindowUpdate updatewindow]
    menu   = Menu "File"
                 [MenuItem "Run"  [MenuId        mId,
                                   MenuShortKey  'r',
                                   MenuFunction  (run ids)],
                  MenuSeparator,
                  MenuItem "Quit" [MenuKey 'q',  MenuFunction quit]] []
    timer  = Timer minute         [TimerId       tId,
                                   TimerSelect   Unable,
                                   TimerFunction (end ids)]
    minute = 60*TicksPerSecond

run :: (Id,Id,Id) *(PState Local) -> *(PState Local)
run (wId,mId,tId) ps
=   updatewindow [TypeDomain]
       {ps & pLocal   = initLocal,
             pIOState = seq [DisableMenuItems [mId],
                             EnableTimers     [tId],
                             EnableKeys       [wId]] ps.pIOState }

quit :: *(PState Local) -> *(PState Local)
quit ps = seqPIO [QuitIO] ps

end :: (Id,Id,Id) TimerState *(PState Local) -> *(PState Local)
end (wId,mId,tId) _ ps=:{pLocal=l}
=   seqPIO [EnableMenuItems  [mId],
            DisableTimers    [tId],
            DisableKeys      [wId],
            DrawInWindow wId [DrawString (toString l.count)]] ps

handleKeys :: Id KeyState *(PState Local) -> *(PState Local)
handleKeys wId (key,KeyDown,_) ps=:{pLocal=l}
=   seqPIO [DrawInWindow wId draw] {ps & pLocal={l & lines=text1,
                                                     count=l.count+1}}
where
    (text1,draw)  = addCharToText key l.lines
handleKeys _ _ ps = ps
```

Figure 3.9 A simple typist monitoring process. Lists with head item `x` and tail list `xs` are denoted by `[x:xs]`. Anonymous variables can be denoted with the wildcard symbol _. `TicksPerSecond` is a system dependent constant for a time interval of one second.

3.3 Dynamic process creation

In this section we describe how to handle dynamic creation of interactive processes. Every interactive process can spawn interactive processes which will run interleaved with their parent process. Every new interactive process runs independently of its parent process, i.e. termination of the parent process has no consequences for the child process, and vice versa. The interleaving of interactive processes is such that abstract event handlers are atomic operations. At this stage interactive processes can communicate by means of file access.

3.3.1 The programmer's view

In Section 3.2 we have described how interactive processes are defined by abstract device definitions and a program state. So interactive processes that spawn new interactive processes need to provide the system with the abstract device definitions and program state of the new process. For this purpose the function `NewIO` is provided (see Figure 3.10).

```
NewIO :: (InitIO *(PState .l)) .l *(IOState .m) -> *(IOState .m)
```

Figure 3.10 The type of the process creation function `NewIO`.

The type of `NewIO` is very similar to the type of `OpenIO`. The difference is that during evaluation of interactive processes the system view of programs is given by `PState` environments rather than the `World` environment which exists only outside interactive processes. `NewIO` creates the new interactive process and takes care that the new interactive process joins the evaluation of interactive processes in an interleaved way. The interleaving is such that event handlers can be regarded as atomic operations. The type of `NewIO` expresses that the program state type `l` of the child process is allowed to differ from the program state type `m` of the parent process.

3.3.2 The system view

The system of Section 3.2 is now faced with two complications when handling an arbitrary number of interactive processes dynamically:

(**a**) how to store processes of different type, and

(**b**) how to share unique global environments between processes.

(**a**) The type of `NewIO` allows the program states of interactive processes to be different. Straightforward extension of the `GState` record with a list of processes of type `[Process l]` enforces every process to have a program state of the *same type*, so this is inadequate. In principle it is possible to unify these types in a functional language with a type system as Clean's but this is a complicated solution (see Chapter 5).

An alternative solution is given by *existential types* (Mitchell and Plotkin, 1985; 1988). In Figure 3.11 the type `Process E.l` is an *existential type* of which the type variable `l` is *existentially quantified*. In Clean a type definition can hide a type variable from its context by prefixing the existential quantifier `E.` to the type variable. The scope of the type variable is limited to the right-hand side of the type. The special type constructor `Void` is used as a type instance of existentially quantified type parameters. As a result, the type `[Process Void]` is a list of processes that can have program states of different types. See Plasmeijer and van Eekelen (1995) for a further account of existential types in the Clean type system. For reasons of event dispatching explained below, the `gProcesses` field, and the new `ioProcesses` field in the `IOState` implementation are a pair of lists (of synonym type `RoundRobin`), rather than one list of processes.

```
::  *GState
=   { gFiles      :: *Files,
      gEvents     :: *Events,
      gProcesses :: RoundRobin (*Process Void) }
```

```
::  *Process E.l
=   { ppLocal    :: l,
      pId        :: *Id,
      pInitIO    :: InitIO       *(PState l),
      pDevices   :: [DeviceState *(PState l)]  }
::  *IOState l
=   { ioDevices  :: [DeviceState *(PState l)],
      ioInitIO   :: InitIO       *(PState l),
      ioId       :: *Id,
      ioEvents   :: *Events,
      ioProcesses:: RoundRobin (*Process Void) }
::  RoundRobin x
:== ([x],[x])

doneRR :: (RoundRobin .x) -> Bool
doneRR (_,todo) = todo==[]

setRR  :: (RoundRobin .x) -> (RoundRobin .x)
setRR  (done,todo)
|   todo==[]  = (todo,done)
|   otherwise = (done,todo)

topRR :: (RoundRobin .x) -> (.x, RoundRobin .x)
topRR (done,[x:todo]) = (x, (done,todo))

holdRR :: .x (RoundRobin .x) -> RoundRobin .x
holdRR x (done,todo) = (done,[x:todo])

insertRR :: .x (RoundRobin .x) -> RoundRobin .x
insertRR x (done,todo) = (done++[x],todo)
```

Figure 3.11 The process administration types, and five access operations on `RoundRobin` data structures. The elements that have been handled are collected in the first list, and the other elements are collected in the second list. The function `doneRR` tests whether all elements have been handled, `setRR` swaps the lists in case all elements have been handled, `topRR` yields the first element that needs to be handled, `holdRR` keeps an element in the second list, and `insertRR` appends a handled element to the first list.

(**b**) The state changed by interactive processes is the process state, and therefore their states require the global environments `Files` and `Events`. However, these environments are unique and can therefore not be shared between interactive processes. The solution to this problem is to pass around the unique environments in the event loop to every process for which we compute a new process state. This is actually already done in the event loop of Section 3.2.2(**c**) for one process by the functions `connect` and `disconnect`. The new event loop, the *interleaved event loop*, applies events to a list of processes rather than one process and also passes the global environments to each process. Terminated processes are removed from the list of processes. This is taken care of by the new definition of `disconnect`. As a result, the interleaved event loop terminates only if all interactive processes have terminated. (It should be observed that the type of `connect`, as presented in Figure 3.12, is not correct because of the occurrence of an existentially quantified variable in its result type (the field `pLocal`). This can be solved by placing the function body of `connect` into the function body of `dispatch`. However, for reasons of presentation `connect` is presented as a distinct function.)

```
connect :: *GState -> *(PState .l)
connect {gFiles,gEvents,gProcesses}
=   { pLocal   = ppLocal,
      pFiles   = gFiles,
      pIOState = { ioDevices   = pDevices,
                   ioInitIO    = pInitIO,
                   ioId        = pId,
                   ioEvents    = gEvents,
                   ioProcesses = procs }}
where
    ({ppLocal,pInitIO,pDevices,pId},procs) = topRR gProcesses

disconnect :: *(PState .l) -> *GState
disconnect {pLocal,pFiles,pIOState={ioDevices,
                                    ioInitIO,
                                    ioId,
                                    ioEvents,
                                    ioProcesses}}
=   { gFiles          = pFiles,
      gEvents         = ioEvents,
      gProcesses      = procs    }
where
    procs
    |   ioDevices==[] = ioProcesses
    |   otherwise     = insertRR p ioProcesses
    p   = { ppLocal   = pLocal,
            pInitIO   = ioInitIO,
            pDevices  = ioDevices,
            pId       = ioId     }

OpenIO :: (InitIO *(PState .l)) .l *World -> *World
OpenIO initIO local world
=     closeworld (gsn.gFiles,gsn.gEvents)
where (fs,es) = openworld world
      proc    = { ppLocal    = local,
                  pId        = InitId,
                  pInitIO    = initIO,
                  pDevices   = []          }
      gs0     = { gFiles     = fs,
                  gEvents    = es,
                  gProcesses = ([],[proc]) }
      gsn     = loop emptyGS nextGS gs0

      emptyGS gs=:{gProcesses=([],[])} = (True, gs)
      emptyGS gs                       = (False,gs)

      nextGS  gs
      =     loop allprocessesdone (dispatch e) gs2
      where (e, es1) = nextEvent gs.gEvents
            gs1      = {gs & gsEvents=es1}
            gs2      = {gs1 & gProcesses=setRR gs1.gProcesses}

            allprocessesdone gs=:{gProcesses=(_,[])} = (True, gs)
            allprocessesdone gs                      = (False,gs)
```

Figure 3.12 The interleaved event loop. The definition of `dispatch` does not change.

The interleaved event loop is an *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*.

Finally, Figure 3.13 shows the definition of `NewIO`. `NewIO` creates the process record and inserts it into the process list of its `IOState` argument, so that next time the new process is scheduled for evaluation.

```
NewIO :: (InitIO *(PState .l)) .l *(IOState .m) -> *(IOState .m)
NewIO initIO local io
=     io2
where ([newid:_],io1) = IOGetIds io
      proc = {ppLocal=local, pId=newid, pInitIO=initIO, pDevices=[]}
      io2  = {io1 & ioProcesses=insertRR proc io1.ioProcesses}
```

Figure 3.13 The definition of `NewIO`.

3.4 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, interactive processes that share a data structure can communicate by writing and reading the shared data structure. Therefore this kind of inter-process communication is called *data sharing*. Because abstract event handlers are atomic operations access to the shared data structure is also atomic.

3.4.1 The programmer's view

Every interactive process defines a public component of arbitrary type in its process state that is used as the shared data structure (Figure 3.14). So the process state of every interactive process consists of a *local* component and a *public* component. For the public component we use the type variables `p` and `q` in (function) type definitions. The types of the functions `OpenIO` and `NewIO` change in a non-essential way.

```
::  *PState l p
=   { pLocal   :: l,
      pPublic  :: p,
      pFiles   :: *Files,
      pIOState :: *(IOState l p) }

OpenIO ::(InitIO *(PState .l .p)) (.l,.p) *World -> *World
NewIO  ::(InitIO *(PState .l .p)) (.l,.p) *(IOState .m .q)
                                       -> *(IOState .m .q)
```

```
ShareIO::(InitIO *(PState .l .p)) .l       *(IOState .m .p)
                                        -> *(IOState .m .p)
```

Figure 3.14 The programmer's view.

An interactive process can spawn a so called *shared* interactive process with the 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 the parent and child process must be equal. The public component will be shared during evaluation of both processes. Every shared interactive process that is spawned by any interactive process shares the public component of its parent process. These processes form a *group of interactive processes*. It should be observed that `ShareIO` does not need to define an initial value of type `p` for the public component because this value already exists.

3.4.2 The system view

The extension of the 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 is that the new event loop does not traverse a list of processes but a list of groups of processes. A group of processes is a record of the shared public data structure and the list of processes that share this structure.

Figure 3.15 presents the internal data type definitions, and Figure 3.16 presents the new definitions of `connect`, `disconnect`, and `OpenIO` which constitute the data sharing interleaved event loop. The same proviso with respect to the type of `connect` in Section 3.3.2(**b**) holds here as well. The definition of `ShareIO` is the same as `NewIO` in Section 3.3 except that the newly created `Process` record is added to the `ioGroup` field of the `IOState` argument. The new definition of `NewIO` adds a `Group` record to the `ioGroups` field of the `IOState`.

```
::  *GState
=   { gFiles     :: *Files,
      gEvents    :: *Events,
      gGroups    :: RoundRobin *(Group   Void)  }
::  *Group    E.p
=   { gPublic    :: p,
      gProcesses:: RoundRobin *(Process Void p)}
::  *Process  E.l p
=   { ppLocal    :: l,
      pId        :: *Id,
      pInitIO    :: InitIO      *(PState l p),
      pDevices   :: [DeviceState *(PState l p)] }
::  *IOState l p
=   { ioDevices :: [DeviceState *(PState l p)],
      ioInitIO   :: InitIO      *(PState l p),
      ioId       :: *Id,
      ioEvents   :: *Events,
      ioGroup    :: RoundRobin *(Process Void p),
      ioGroups   :: RoundRobin *(Group   Void)  }
```

Figure 3.15 The data sharing administration types.

```
connect :: *GState -> *(PState .l .p)
connect {gFiles,gEvents,gGroups}
=   { pLocal   = ppLocal,
      pPublic  = gPublic,
      pFiles   = gFiles,
      pIOState = { ioDevices = pDevices,
                   ioInitIO  = pInitIO,
                   ioId      = pId,
                   ioEvents  = gEvents,
                   ioGroup   = ps,
                   ioGroups  = gs }}
where
    ({gPublic,gProcesses},gs)           = topRR gGroups
    ({ppLocal,pId,pInitIO,pDevices},ps) = topRR gProcesses

disconnect :: *(PState .l .p) -> *GState
disconnect {pLocal,pPublic,pFiles,pIOState={ioDevices,
                                            ioInitIO,
                                            ioId,
                                            ioEvents,
                                            ioGroup,
                                            ioGroups}}
=   { gFiles  = pFiles,
      gEvents = ioEvents,
      gGroups = groups  }
where
    proc = { ppLocal   = pLocal,
             pInitIO   = ioInitIO,
             pDevices  = ioDevices,
             pId       = ioId     }
    procs
    |   ioDevices==[]  = ioGroup
    |   otherwise      = insertRR p ioGroup
    groups
    |   procs==([],[]) = ioGroups
    |   doneRR procs   = insertRR {gPublic    = pPublic,
                                   gProcesses = setRR procs} ioGroups
    |   otherwise      = holdRR   {gPublic    = pPublic,
                                   gProcesses = procs      } ioGroups

OpenIO :: (InitIO *(PState .l .p)) (.l,.p) *World -> *World
OpenIO initIO (local,public) world
=     closeworld (gsn.gFiles,gsn.gEvents)
where (fs,es) = openworld world
      proc    = { ppLocal    = local,
                  pId        = InitId,
                  pInitIO    = initIO,
                  pDevices   = []           }
      group   = { gPublic    = public,
                  gProcesses = ([],[proc])  }
      gs0     = { gFiles     = fs,
                  gEvents    = es,
                  gGroups    = ([],[group]) }
      gsn     = loop emptyGS nextGS gs0

      emptyGS gs=:{gGroups=([],[])} = (True, gs)
      emptyGS gs                    = (False,gs)
```

```
nextGS  gs
=      loop allgroupsdone (dispatch e) gs2
where (e, es1) = nextEvent gs.gEvents
      gs1      = {gs & gsEvents=es1}
      gs2      = {gs1 & gProcesses=setRR gs1.gGroups}

      allgroupsdone gs=:{gGroups=(_,[{gProcesses=(_,[])}])}
                                   = (True, gs)
      allgroupsdone gs=:{gGroups=(_,[])} = (True, gs)
      allgroupsdone gs                   = (False,gs)
```

Figure 3.16 The data sharing interleaved event loop. The definition of `dispatch` does not change.

3.5 Message passing

In this section we introduce the message passing mechanism of the interleaved event I/O system. Interactive processes can send messages to any other interactive process by either *synchronous* or *asynchronous* message passing. In our system messages are considered to be abstract events. To conform with 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*. Receivers can be created and disposed of dynamically. Message passing is polymorphic: 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 program to send messages of the wrong type.

3.5.1 The programmer's view

Figure 3.17 presents the definition of the abstract receiver device. Analogous to the definition of abstract devices in the I/O system, the receiver device is defined by an algebraic data type, `ReceiverDef`. 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`.

```
::  ReceiverDef      m ps
=   Receiver (ReceiverFunction m ps) [ReceiverAttribute ps]
::  ReceiverFunction m ps
:== m ps -> ps
::  ReceiverAttribute  ps       // Default:
=   ReceiverSelect SelectState  // Able
::  RId m

OpenReceiver :: (ReceiverDef m *(PState .l .p)) *(IOState .l .p)
             -> (RId m,  * IOState .l .p)
CloseReceiver:: (RId m)  *(IOState .l .p) -> *IOState .l .p

ASyncSend    :: (RId m) m *(PState .l .p) -> * PState .l .p
SyncSend     :: (RId m) m *(PState .l .p) -> * PState .l .p
```

Figure 3.17 The programmer's view of the receiver device.

Interactive processes can dynamically open and close an arbitrary number of receivers with the functions `OpenReceiver` and `CloseReceiver`. In order to identify the addressee of a message uniquely, `OpenReceiver` generates a fresh, abstract identification value of type `RId m` for each receiver in the same sense as `IOGetIds` generates fresh identification values for general purposes. The crucial difference with values of type `Id` is that receiver identifications are parameterized with the message type of the corresponding receiver function. It should be observed that interactive processes can create receivers of different message type.

Interactive processes can send messages to arbitrary interactive processes by asynchronous and synchronous message passing (with `ASyncSend` and `SyncSend` respectively). Both functions require the identification value of type `RId m` of the receiver and a message of corresponding type `m`. Neither function has an effect in case the receiver does not exist anymore. `ASyncSend` is purely asynchronous. Using `SyncSend` the sending interactive process blocks until the indicated receiver accepts the message.

3.5.2 The system view

In this section we will show how the extended system handles receivers of arbitrary message types dynamically and asynchronous and synchronous message passing respectively. Finally, we show how type-safe message passing is handled.

3.5.2.1 Receiver administration

Figure 3.18 gives the administration types needed to accomodate the new abstract device. Recall that the run-time administration of devices is of type `DeviceState` environment. The run-time state of a receiver is the record type `RState`, which consists of a local message queue (initially empty), a receiver function, and the current `SelectState` (initially **`Able`** in case no `ReceiverSelect` attribute is given).

```
::  DeviceState ps
=   ReceiverState [HiddenRState] | …

::  RState m ps
=   { rMessQ    :: [m],
      rFunction :: m -> ps -> ps,
      rSelect   :: SelectState}

rOpen :: (ReceiverDef m .ps) -> RState m .ps
rOpen (Receiver f [])
    =   {rMessQ=[], rFunction=f, rSelect=Able}
rOpen (Receiver f [ReceiverSelect s : _])
    =   {rMessQ=[], rFunction=f, rSelect=s}

rAddMessage :: m (RState m .ps) -> RState m .ps
rAddMessage m rState = {rState & rMessQ=rState.rMessQ++[m]}

rGetFunction :: (RState m .ps) -> ReceiverFunction m .ps
rGetFunction {rSelect=Able, rFunction=f} = f
rGetFunction _                           = I
```

Figure 3.18 The receiver device administration and receiver state access operations. Function `++` is list concatenation, and `I` is the identity function.

It should be noted that it is not adequate for `DeviceState` to administrate its receivers as `[RState m ps]` because this forces different receivers to act on the same type of message. We delay the discussion how `OpenReceiver` actually stores receivers until Section 3.5.2.4. From Figure 3.18 one can conclude that the `DeviceState` type not only hides the message type of its receivers but also the process state type, as it is defined on `[HiddenRState]`.

3.5.2.2 Asynchronous message passing

In order to obtain asynchronous message passing, it is sufficient for `ASyncSend` to add the message to the local message queue of the proper receiver. This can be done because the `PState` record argument of `ASyncSend` contains all currently existing processes. So, basically `ASyncSend` locates the proper `RState` record in the existing processes, and adds the message to the record by applying `rAddMessage` (Figure 3.18). If the record is not found then `ASyncSend` has no effect.

As a result, sending a message to a receiver that is still open at the time the message is being sent, adds the message to its message queue (regardless whether the receiver is in enabled or disabled state). Furthermore, messages are handled in the same order as they are sent.

The receiver device, when evaluated (which is some time after evaluation of `ASyncSend`), retrieves the message `m` from its local message queue and applies its receiver function `f` to the message and the current process state `ps`, so the response is `f m ps`.

3.5.2.3 Synchronous message passing

In a synchronous message passing scheme the sending process blocks evaluation until the receiver has actually accepted the message. Sending a message occurs in the course of evaluation of an abstract event handler. Consequently the event handler cannot be completely evaluated because the receiver must first accept the message. In our scheme accepting a message means evaluation of the corresponding receiver function. So we must evaluate the proper receiver function. Because this function must be applied to the process state of the receiving process (which is in general different from the sending process) `SyncSend` needs to do a *temporary context switch*.

Figure 3.19 shows how `SyncSend` performs the context switch. First the current `PState` record is mapped to the `GState` record with `disconnect` (Section 3.2.2). From the `GState` record `locateR` determines which interactive process contains the receiver to which the message should be sent. The corresponding `PState` record of that process is created with `connect`. From the `PState` record `getReceiverF` retrieves the receiver function. Basically, `getReceiverF` applies `rGetFunction` (Figure 3.18) to the specific `RState` record of its `PState` argument identified by the `RId` argument. The receiver function it has retrieved is then applied to the message and the `PState` record. The new `PState` record is then disconnected and transformed back into the original `PState` record after which evaluation of the blocked abstract event handler continues. It should be noted that in this scheme synchronous message handling can overtake asynchronous message handling.

```
SyncSend :: (RId m) m *(PState .l .p) -> *(PState .l .p)
SyncSend rId message pState
=   connect (i, gState1)
where
    (i,gState)   = disconnect pState
    pState1      = connect (locateR rId gState)
    (f,pState2) = getReceiverF rId pState1
```

```
        pState3      = f message pState2
        (_,gState1) = disconnect pState3
```

Figure 3.19 Synchronous message passing.

3.5.2.4 Type security

In Section 3.5.2.1 we argued that in order to accomodate receivers of different message types polymorphic types are inadequate. In Section 3.3.2 we have shown how existential types can be used to hide such type information. However, existential types limit all access to an existentially quantified type within the scope of the existential quantification. But this is to restrictive in the case of asynchronous message passing (updates the receivers message queue) and in case of synchronous message passing (reads the receiver function).

In this section we introduce a special abstract type, `Hidden`, that will be used as a black box in which arbitrary data structures can be hidden. One other special abstract type, `HiddenId x`, is introduced that will be used to access data structures of type `x` in a `Hidden` object. We show how we can break the type abstraction created by existential types and obtain a type-safe update and type-safe read operation. Then we show how receivers can be stored and accessed safely as a special case of this scheme.

Figure 3.20 gives the definition of the abstract type `Hidden`. The only way instances of type `Hidden` and `HiddenId` are created is by `hide`, which hides any object of some type `x`. The unique `Id` argument value is used to generate a fresh `HiddenId x` value. So the `HiddenId` value unambiguously identifies the `Hidden` object created by `hide`. This value is the key used by the two access functions `writeHidden` and `readHidden` to open the type abstraction enforced by the existential quantification.

Let `write`::`x`→`x` be a function that updates a value of type `x`. Then, given this function and a `HiddenId x` value, `writeHidden` updates a `Hidden` value only if the `HiddenId` values coincide, in which case the `Hidden` record is known to contain a data structure of type `x`.

Let `read`::`x`→`y` be a function that reads a value of type `x` as a value of type `y`. Then, given this function and a `HiddenId x` value, `readHidden` reads a `Hidden` value only if the `HiddenId` values coincide, in which case the `Hidden` record is known to contain a data structure of type `x`.

Both access functions are still ill-typed. However, because of abstract types and the generation of fresh identification values it can be shown that assuming these types are correct every well-typed program has type-safe access to `Hidden` objects. So the type checker can safely assume the given types of `writeHidden` and `readHidden`. Because Clean programmers work in a strongly typed system, they use a well-typed library. Internally, type verification is disabled locally for these two specific operations only.

```
::  Hidden       :== Hidden` Void
::  Hidden` E.x =   { hiddenId :: HiddenId x,
                      hidden   :: x}
::  HiddenId x  :== Id
::  Optional x  =   One x | None

hide :: *Id x -> (*Id, HiddenId x, Hidden)
hide id x
=   (id`,xId,hide_it)
where
```

```
    ([xId:_],id`) = GenIds id
    hide_it       = {hiddenId=xId, hidden=x}

writeHidden :: (x -> x) (HiddenId x) Hidden -> Optional Hidden
writeHidden write id hide
|   id == hide.hiddenId = One {hide & hidden=write hide.hidden}
|   otherwise           = None

readHidden :: (x -> y) (HiddenId x) Hidden -> Optional y
readHidden read id hide
|  id == hide.hiddenId  = One (read hide.hidden)
|  otherwise            = None
```

Figure 3.20 The abstract type `Hidden` for type guarded access to hidden data structures.

It is easy to see how this scheme can be applied to hide arbitrary `RState` records. First, the type `HiddenRState` is a synonym type of `Hidden`. `OpenReceiver` applies `hide` to a `RState` record created by `rOpen` and adds it to the list of receivers. `ASyncSend` applies guarded update access with `rAddMessage` in order to add the message to the message queue of the indicated receiver. Finally, `SyncSend` applies guarded read access with `rGetFunction` in order to retrieve the receiver function.

3.6 Example

In this section we present an example of how an interactive graphical user interface program can be constructed with the primitives that have been discussed in the previous sections. It is an extended version of the program presented at the end of Section 3.2 (Figure 3.9). In this version the program is constructed of two distinct interactive processes: the typist process and the monitor process. There is a one-directional asynchronous message passing communication from the typist process to the monitor process. Some minor functions and constants have been omitted in the program code for reasons of clarity. Figure 3.21 gives a snapshot of the application running on a Macintosh system.

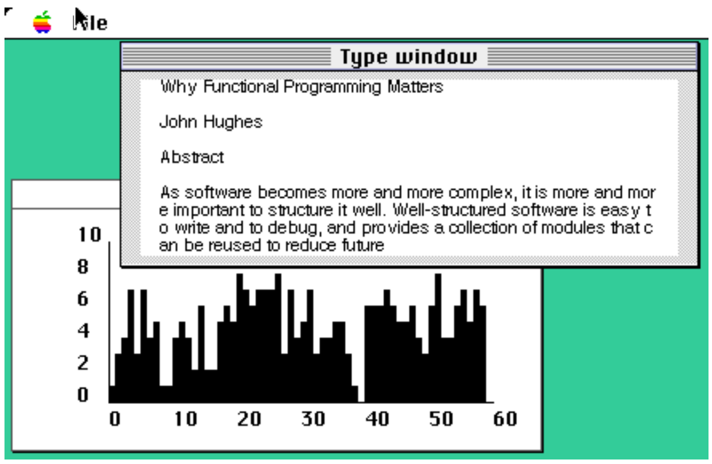


Figure 3.21 A snapshot of the typing monitor running.

The monitor process (Figure 3.22) 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 a type variable. 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 the initial interactive process created by the program. At start it creates its timer, window, and receiver. Then it creates the typist process parameterized with the monitor receiver address. The monitor process is controlled by the messages of type `Message` it receives from the typist process. This is shown clearly by the receiver function `receive`. If it receives the **`Start`** message, it sets the local process state to its initial value and enables the timer. It disables the timer if it receives **`End`**. For each **`KeyHit`** it increments `count`. Finally, on acceptence of **`Quit`** it quits the monitor process. The timer event handler `drawKeyHits`, when enabled, is evaluated once every second. It appends the current `count` to `counts`, resets `count` to zero, increments `time`, and draws the current `count` in its window.

```
::  Message   =   Start | End | KeyHit | Quit
::  Local     =   { count  :: Int,
                    counts :: [Int],
                    time   :: Int  }
::  MState p :== PState Local p

Start :: *World -> *World
Start world = OpenIO [initIO] (initLocal, Nil) world

initLocal :: Local
initLocal = {count=0, counts=[], time=0}

initIO :: *(MState .p) -> *MState .p
initIO ps
=   openTypist talkTo {ps & pIOState=io1}
where
    ([wId,tId:_],io ) = IOGetIds ps.pIOState
    (talkTo,     io1) = OpenReceiver   receiver (
                        OpenWindow     window   (
                        OpenTimer      timer)) io
    receiver = Receiver [ReceiverFunction (receive (wId,tId))]
    window   = Window "Monitor" MonitorDomain []
                        [WindowId      wId,
                         WindowUpdate  updatemonitor]
    timer    = Timer Second
                        [TimerId       tId,
                         TimerFunction (drawKeyHits wId)
                         TimerSelect   Unable]

receive :: (Id,Id) Message *(MState .p) -> *MState .p
receive (wId,tId) Start ps=:{pIOState=io}
=   updatemonitor [MonitorDomain]
        (seqPIO [EnableTimers [tId]] {ps & pLocal=initLocal})
receive _ KeyHit ps=:{pLocal=local}
=   {ps & pLocal={local & count=local.count+1}}
receive (_, tId) End ps
=   seqPIO [DisableTimers [tId]] ps
receive _ Quit ps
=   seqPIO [QuitIO] ps

drawKeyHits :: Id TimerState *(MState .p) -> *MState .p
drawKeyHits wId _ ps=:{pLocal  ={count=c,counts=cs,time=t},
```

```
                         pIOState=io}
=    {ps & pLocal   = {count=0,counts=append c cs,time=t+1},
           pIOState = DrawInWindow wId [drawCount t c] io}
```

Figure 3.22 The main code of the monitor process.

The typist process is almost identical to the process presented at the end of Section 3.2 (Figure 3.9). The difference is that it does not count key hits but instead controls the monitor process. Therefore the `count` field of the local program state record `Local` is obsolete. Because it is not the initial interactive process, the function `openTypist`, which spawns the typist process, replaces the `OpenIO` call with `NewIO`. In addition to its previous actions `Run` sends the **`Start`** message to the monitor process. The timer function `end` sends **`End`** to the monitor process. During a session, `handleKeys` sends **`KeyHit`** to the monitor process for every key that has been pressed. `Quit` sends **`Quit`** to the monitor process and terminates the typist process. The changes to the code of Figure 3.9 are listed in Figure 3.23.

```
::  Local    :== { lines :: [String] }
::  TState p :== PState Local p

openTypist :: (RId Message) *(PState .l .p) -> *(PState .l .p)
openTypist monitorid ps
=   seqPIO [NewIO [initIO monitorid] (initLocal,Nil)] ps

initIO :: (RId Message) *(TState .p) -> *(TState .p)
initIO monitorid ps = seqPIO [OpenTimer  timer,
                              OpenWindow window,
                              OpenMenu 0 menu] ps
where
   …
   window = Window "Type window" TypeDomain []
              [WindowKeys Able (handleKeys` monitorid wId),…]
   menu   = Menu "File"
              [MenuItem "Run"  [… ,MenuFunction  (run`  monitorid ids)],
               MenuSeparator,
               MenuItem "Quit" [… ,MenuFunction  (quit` monitorid)]] []
   timer  = Timer minute       [… ,TimerFunction (end`  monitorid ids)]

run` :: (RId Message) (Id,Id,Id) *(TState .p) -> *(TState .p)
run` monitorid ids ps = ASyncSend monitorid Start (run ids ps)

quit` :: (RId Message) *(TState .p) -> *(TState .p)
quit` monitorid ps = ASyncSend monitorid Quit (quit ps)

end` :: (RId Message) (Id,Id,Id) TimerState *(TState .p)
                                          -> *(TState .p)
end` monitorid ids t ps = ASyncSend monitorid End (end ids t ps)

handleKeys` :: (RId Message) Id KeyState *(TState .p) -> *(TState .p)
handleKeys` monitorid wId (key,KeyDown,_) ps=:{pLocal=l}
=  seqPIO [ASyncSend monitorid KeyHit,
           DrawInWindow wId draw] {ps & pLocal={l & lines=text1}}
where
   (text1,draw)  = addCharToText key l.lines
handleKeys` _ _ _ ps = ps
```

Figure 3.23 The changes required for the program code of Figure 3.9.

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.

3.7 Related work

The research described in this chapter 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 in the FUDGETS system dynamic process creation is very limited. A comparison between the FUDGETS system and the Clean event I/O system is given by Noble and Runciman (1994).

The area of functional operating systems offers more closely related work with respect to dynamic process creation and inter-process communication. One particular system is the Kent Applicative Operating System (KAOS) project (Turner, 1990). The system is based on earlier work by Stoye (1984). Both systems allow dynamic creation of functional processes. The inter-process communication by message passing 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. The idea of type-safe message passing by means of type parameterized receiver identification values was inspired by the KAOS approach.

3.8 Conclusions

In this chapter we have presented an interleaved state transition system (the interleaved event I/O system) that is dynamically composed of single state transition systems (interactive processes). We have defined two forms of inter-process communication in this system, namely data sharing, and asynchronous and synchronous message passing. Both forms of communication are type-safe and polymorphic. The construction is done entirely in a pure functional framework. The interleaved event I/O system can be regarded as an operational semantics of a mini operating system allowing dynamic interleaved interactive functional processes.

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. 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 and water level of a simulated water tank.

3.9 Current and future work

As stated in the introduction the interleaved event I/O system has not been introduced to study concurrent interactive processes. Our ultimate interest however is an event I/O system in which one can program distributed graphical user I/O programs. Currently, we are studying how to set up the system that will allow interactive processes to be evaluated concurrently and distributed over a network of processors.

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 chapter.