4
Concurrent Interactive Functional Processes

In this chapter we present an operational semantics for *concurrent interactive processes* in the purely functional programming language Clean. An interactive process is in essence a state transition system which apart from its logical state can also access the state of the file system, do high-level graphical user I/O, and do inter-process communication via synchronous and asynchronous message passing and data sharing. Inter-process communication is type-safe and polymorphic. The semantics of the system is based on earlier work on the *interleaved event I/O system*. In this chapter we identify limitations of the interleaved model in the context of concurrent processes and propose a new process semantics that does allow a concurrent implementation. The method basically introduces a remote procedure call communication scheme and demonstrates how to apply this scheme to obtain a concurrent process semantics. The resulting system is the concurrent event I/O system. The operational semantics is given in Clean itself. As a result the concurrency system is completely functional because the construction is done within the pure functional framework.

4.1 Introduction

This chapter reports part of a research project conducted on the practical application of pure functional programming languages. Some of the research topics are destructive updates of data structures, graphical user interface programming, parallel process creation and management, forms of inter-process communication, and efficient implementations. In this chapter we discuss how to obtain concurrent processes that themselves engage in graphical user interface and file I/O in a pure functional framework.

The lazy, purely functional programming language Clean offers a library (written entirely in Clean), the *event I/O system*, to program complex graphical user interface applications on a high level of abstraction (Chapter 2, and Achten *et al.*, 1993). In essence an event I/O program is a structured set of higher-order functions (called *abstract event handlers*) that define what graphical user interface elements the program uses (such as menus, windows, dialogues, and timers) and also the response of the program to *abstract events* (such as selection of menu items, mouse and keyboard actions). The set of functions is structured by means of algebraic data types which are abstract definitions of graphical user interface elements. For more details see Chapter 2. Abstract event handlers are *state transition functions*. The *state* of an event I/O program can be of arbitrary type. The semantics of the event I/O system is a state transition semantics. Given the initial state of an event I/O program and an initial set of abstract event handlers (both provided by the programmer), the system evaluates for each new abstract event the corresponding abstract event handler. This yields a new state value. This is repeated until termination of the program.

The interleaved event I/O system is the extension of the event I/O system with *interactive processes* (Chapter 3). Interactive processes are the conceptual units by which Clean programmers can construct more complex interactive programs from simpler interactive programs. The main difference with the event I/O system is that these processes coexist in an interleaved fashion, hence the name of the system. Interactive processes can be created and destroyed dynamically. In the interleaved event I/O system interactive processes are defined in the same way as interactive programs are in the event I/O system; by means of structured sets of abstract event handlers. Each interactive process has a private state of arbitrary type. Inter-process communication primitives that have been defined in the interleaved system are data sharing and (a)synchronous message passing. These methods of communication are polymorphic and type-safe. The major semantic challenge of the interleaved event I/O system was to solve how independent interactive processes can use the same file system, share global data, and use message passing in a pure functional framework. In essence the semantics of the interleaved event I/O system is based on the atomic, interleaved evaluation of process state transition functions. This process state consists of a local part, containing the private state information of the process, and a global part, containing the file system and global information. Section 4.2 gives a more detailed overview how this is accomplished.

The interactive process concept in the interleaved event I/O system is well-suited to explain the behaviour of a program that is dynamically composed of interactive processes. However, the semantics cannot without modification be used to explain the semantics of true parallel evaluation of interactive processes. We are interested in such a concurrency model because we intend to program real distributed interactive applications using the graphical user interface tools from the event I/O system.

To illustrate the limitations of the interleaved model with respect to concurrency consider the following case. Let program *P* consist of the interactive processes *A* and *B*. Let f_1 and f_2 denote initially closed files. *A* and *B* both have the same state transition function that can be evaluated repeatedly. This function does the following: subsequently open f_1 and f_2; if either operation fails because the file was already open then terminate A and B, otherwise close f_1 and f_2.

In the interleaved semantics P does not terminate. Consider the base case in which files f_1 and f_2 are closed and that either one of the transition functions of A and B is chosen. Because in the interleaved model transition function are evaluated completely both files have been opened and closed successfully after evaluation of the transition function. Consequently after evaluation of either function we have the base case again. So P never terminates.

If one interprets this case from a concurrent point of view in which transition functions are evaluated in parallel rather than atomically, then P can terminate. One of the scenarios in which this occurs is that either A or B has just opened f_1, after which the other process also tries to open f_1. For the latter process the operation fails, so both processes are terminated. The interleaved semantics does not catch the situation which might arise in a parallel setting simply because the granularity of interleaving is unrealistically coarse.

In a real distributed implementation interactive processes are implemented as concurrent reduction processes. Implementations of interactive processes with interleaving semantics based on reduction processes with concurrency semantics give rise to some problems. For instance, in order to prevent processes from opening the same file concurrently one needs to lock the complete file system. This causes unacceptable sequentialization of processes that might well be evaluated concurrently if they operate on disjoint sets of files.

In this chapter we show how to obtain a concurrency model, the *concurrent event I/O system*, from the interleaved event I/O system that allows interactive processes to be evaluated in parallel. The main reason for the interleaved event I/O system to obstruct parallel evaluation of interactive processes is the sequentialization of process state transition functions caused by data dependency created by data sharing. So in essence the solution is to eliminate data sharing. This is done for the data shared file system. Data sharing of arbitrary data structures between processes is not eliminated but is restricted per processor.

The semantics of the concurrent 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), and last but not least it allows specifications to be type-checked, compiled, and tested (as such specifications are readily executable). The specification of the concurrent event I/O system can be used as a framework for the actual implementation.

The concepts that are studied in this chapter are not new: remote procedure calls, (a)synchronous message passing, dynamic process management, and so on are relatively well-known concepts. In this chapter we show that these concepts can be defined in a pure functional framework, maintaining the advantages of functional programming, and increasing expressiveness of these concepts by polymorphism and strong type systems.

This chapter is structured as follows. Section 4.2 gives a brief overview of the interleaved event I/O system. Section 4.3 gives some motivations of how to derive the concurrent event I/O system from the interleaved event I/O system. The technical details are worked out in Sections 4.4 and 4.5. Section 4.4 extends the interleaved system with a new communication primitive for Remote Procedure Calling and remote procedure server processes. These are applied in Section 4.5 to obtain a new definition of the file system. Section 4.6 reconsiders our case example with the new semantics. Section 4.7 gives some implementation considerations of the concurrent system. Related work is presented in Section 4.8 and conclusions are drawn in Section 4.9. Finally we give some leads to current and future work in Section 4.10. Appendix B contains the operational semantics as discussed in this chapter.

4.2 The interleaved event I/O system

In this section we give a very brief overview of the interleaved event I/O system of Clean. This exposition is basically an extended abstract of Chapter 3.

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.

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. See Smetsers *et al.* (1993), Barendsen and Smetsers (1993) for a detailed treatment of the uniqueness type system of Clean.

The interleaved event I/O system provides programs with a *hierarchy* of environments that can be used to do I/O (Figure 4.1). Environments are *independent*: operations on one environment do not affect other environments. 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 (see Figure 4.2). The event stream environment is discussed later.

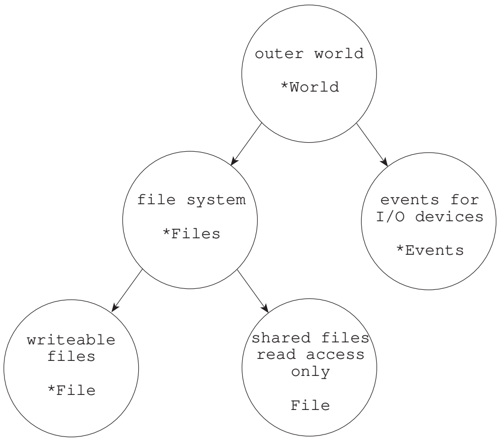

Figure 4.1 The Clean environment hierarchy.

Graphical user interface programs are constructed out of modular components called *interactive processes*. An interactive process basically is a structured set of *process state transition functions*. So if the process state is of type `ps` then the process state transition functions are of type `ps` → `ps`.

```
fopen  :: String Int *Files -> (Bool, *File, *Files)
sfopen :: String Int *Files -> (Bool,  File, *Files)
freopen:: *File  Int *Files -> (Bool, *File, *Files)
fclose :: *File      *Files -> (Bool,        *Files)
stdio  ::            *Files -> (*File,       *Files)

freadc :: *File -> (Bool, Char, *File)
fwritec:: Char *File -> *File
… and other basic types (Int,Real,String)
```

Figure 4.2 The file system operations and some file operations. The Boolean result reports whether the function has successfully opened a file. If not then the (`*`)`File` result is a dummy.

The process state is the predefined parameterized *record type* (`PState l p`) (Figure 4.3) and consists of four components. The first component is the *local process state* of arbitrary type `l` which reflects the logical state of the interactive process. The second component is the *public process state* of arbitrary type `p` which reflects the logical state of the interactive process. For conciseness we use the type variables `l` and `m` for the local type variable and the type variables `p` and `q` for the public type variable in (function) type definitions. The third 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 p
=   { pLocal  :: l,
      pPublic :: p,
      pFiles  :: *Files,
      pIOState:: *IOState l p }
::  InitIO ps
:== [ps -> ps]

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
QuitIO ::        *(IOState .l .p)          -> *IOState .l .p
```

Figure 4.3 The process combinators of the interleaved event I/O system. Type definitions are preceded by ::. Synonym types are distinguished from other type definitions by the :== symbol. The type *list of a* is denoted by *[a]*. The type *function from a to b* is denoted by *a->b*. Type symbols start with a capital, (type) variables always start in lowercase, function names start in either case. Function definitions are optionally preceded by their type definition. An n-ary function named f with arguments of type $\tau_1 \ldots \tau_n$, and result type τ has a type definition f :: $\tau_1\ \tau_2 \ldots \tau_n \rightarrow \tau$.

In Clean, a *record type* is a tuple-like algebraic type with the advantage that selection is done by *field names* instead of position matching. Let `ps` be an expression of type `PState`. On a pattern-match position 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}` is a record equal to `ps` but the field `pFiles` has value `fs`.

The evaluation of interactive processes is done by the library function `OpenIO` (Figure 4.3). `OpenIO` is applied to an initial set of actions (of type `InitIO`), initial values of the local and public process state components, and the `World` environment. It creates the `IOState` environment of the process and evaluates the initial actions in head-to-tail order. These functions can be used by the programmer to open windows, menus, dialogues, timers, and receivers. The process is evaluated until termination and `OpenIO` yields a new `World` value. An interactive process requests termination by applying the library function `QuitIO` to its `IOState` environment.

Every interactive process can spawn an arbitrary number of new interactive processes. This is done with the process combinators `NewIO` and `ShareIO`.

`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 type of `NewIO` expresses that both local and public process state component types `l` and `p` of the child process are allowed to differ from the corresponding component types `m` and `q` of the parent process.

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 *process group*. 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.

Figure 4.4 gives a schematic representation of the structure of an interactive program at run-time. The program is represented by the outer box. Circles represent data structures. The `Files` environment is shared by all process groups. A program consists of a number of process groups each of which shares a common data structure of some type `p`. The process groups are represented by a pile of boxes, which are evaluated interleaved. Each process in a process group has a private process state component of some type `l`. Again, processes are represented by a pile of boxes indicating that they are evaluated interleaved.

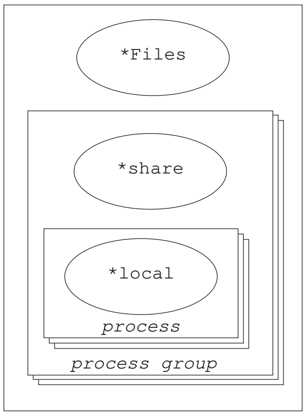

Figure 4.4 Schematic representation of the structure of the interleaved event I/O system.

As mentioned above, interactive processes are structured sets of process state transition functions. Each event retrieved from the event stream environment is dispatched to every interactive process in round-robin order. Every event triggers the evaluation of a well-defined subset of the process's transition functions which are evaluated one-by-one and completely. So process state transition functions are evaluated interleaved and atomically. To evaluate a process state transition function it must be applied to its process state which has to be constructed by the evaluation mechanism. The creation of this process state, and undoing after evaluation, is actually a *context-switch*.

Finally, interactive processes can send messages to any other interactive process by either *synchronous* (`SyncSend`) or *asynchronous* (`ASyncSend`) message passing (Figure 4.5). `ASyncSend` is purely asynchronous. `SyncSend` performs a context-switch to the process with the indicated receiver, handles the message, and performs the context-switch back. Receivers can be created and disposed of dynamically. Creation of a receiver (with `OpenReceiver`) yields a type parameterized identification value of type `RId m`. Message passing is polymorphic: the content of a message can be any typeable expression. Messages can contain in particular higher-order functions, algebraic expressions, and so on. 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.

```
::  ReceiverDef m ps :== m ps -> ps
::  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 4.5 Principal type definitions and functions for message passing of the interleaved event I/O system. Type definitions with a left-hand side only (:: `RId m`) are abstract type definitions. Their actual implementation is hidden from the programmer.

We conclude this section with an example of a program that defines a small window based talk application (Figure 4.6). The program consists of two identical interactive processes. Each process has a window and a receiver. We introduce a synonym type `TalkState` for the process states of the two processes. The local process state is a record consisting of two fields that represent the texts that have been typed by both processes. For simplicity we assume that this is some abstract type `Text` with an operation `addChar` to add a new character to the current text. The function `addChar` yields the new text and a list of drawing functions to give the proper feedback in the display window. The processes do not use data sharing, and therefore the public process state component is a type variable.

Associated with the window is the abstract event handler `sendKeys` which is parameterized with the receiver identification of the other talk process. For each key hit, `sendKeys` adds the hit key to its local process state, draws the hit key in its display window, and asynchronously sends the key to the other talk process. The receiver function of both processes on receiving a new character, adds the character to its local process state, and draws the new key in its display window.

```
::  TalkState p :== PState Local p
::  Local        =   { myText   :: Text,
                       yourText :: Text }
```

```
sendKeys :: (RId Char) KeyState (TalkState .p) -> TalkState .p
sendKeys _ (_,KeyUp,_) ps = ps
sendKeys otherTalkProcess (c,_,_) ps=:{pLocal,pIOState}
=   ASyncSend otherTalkProcess c {ps & pLocal=local1, pIOState=io1}
where
    (drawFs, text1) = addChar c pLocal.myText
    local1          = {pLocal & myText=text1}
    io1             = DrawInActiveWindow drawFs pIOState

receiveKeys :: Char (TalkState .p) -> TalkState .p
receiveKeys c ps=:{pLocal,pIOState}
=   {ps & pLocal=local1, pIOState=io1}
where
    (drawFs, text1) = addChar c pLocal.yourText
    local1          = {pLocal & yourText=text1}
    io1             = DrawInActiveWindow drawFs pIOState
```

Figure 4.6 A program creating two identical processes that communicate by message passing. For clarity of presentation alternative data constructors of algebraic types are printed in **boldface**. The _ symbol is a wildcard for anonymous expressions that are not applied in a function. The `where` clause contains the local definitions that are used in the function. In particular these can also be function definitions.

4.3 Towards the concurrent event I/O system

In Section 4.2 we have seen that the global character of the file system environment is modelled as a shared data structure between all interactive process groups. Because data sharing creates data dependency the interleaved event I/O system is in essence an interleaved system. In order to arrive at a concurrent system we need to find a model for the file system environment that eliminates data dependency. We propose to eliminate data dependency as follows. The file system environment is not shared by every interactive process but becomes the local process state component of a new, globally known, special interactive process, the *file server process*. Interactive processes can request file system services of this process by message passing. This situation is sketched in Figure 4.7. If we compare this scheme to the scheme given in Figure 4.4 we can observe that the file system environment has been moved to the local process state component of the new file server process. In this way data sharing of process groups on the file system has been eliminated. The pile of process group boxes has been replaced by process groups placed in juxtaposition, indicating that these are to be evaluated concurrently.

It is our goal that the file system operations as available to the programmer in the interleaved event I/O system (these have been presented in Figure 4.2) are available still (but for a small change of type) in the concurrent event I/O system even though these may involve interprocess communication. So we have the following requirements the new definitions of the file system functions should meet:

(**a**) Function call blocks the calling interactive process.

(**b**) Function call does not block other interactive processes.

Requirement (**a**) is needed to be able to regard a file system operation as an ordinary function call which yields a result after evaluation. Requirement (**b**) takes care that although communication with the file server is synchronous it takes time for the communication to be handled. Duration of time is defined as the evaluation of other interactive processes during communication.

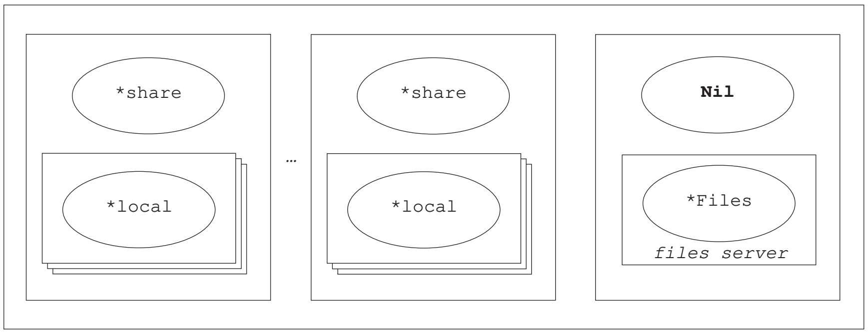

Figure 4.7 Schematic representation of the structure of the concurrent event I/O system.

One of the more generally known communication schemes that satisfies these requirements is the *remote procedure call*. We define this mechanism in our functional framework in Section 4.4. Section 4.5 describes how the file server process is defined and how interactive processes use its services. Given these two extensions we have defined the concurrent event I/O system. Before we continue we make the following remarks:

We intend to obtain a concurrent system by elimination of data sharing of the file system. However, inside process groups we still have a level of data sharing left. We do not eliminate this data sharing because of two reasons: firstly the use of data sharing is, for some problems, the most intuitive solution, and secondly programmers can transform data sharing into a remote procedure call scheme themselves by application of the technique as discussed in the next section.

Finally, it should be observed that individual files belong to the local or public components of interactive processes, so once a process obtains a file this does not cause further sequentialization because environments are by definition independent.

4.4 Remote procedure call processes

In this section we introduce a special kind of interactive process, the *remote procedure call process* (*RPC process*), and its corresponding communication mechanism, the *remote procedure call*. RPC processes are defined by the parameterized record type `RPCDef in out ps`, stating that it is an interactive process that for every message of type `in` generates one message of type `out` (see Figure 4.8). RPC processes are opened with two new process combinators. A new RPC process is either a new process group (`NewRPC`) or a member of the parent process group (`ShareRPC`). Analogous to opening a receiver, the creation of an RPC process yields an identification value which is type parameterized with the message types `in` and `out`. Figure 4.9 gives the definition of `NewRPC` (the definition of `ShareRPC` is given in Appendix B.1).

```
::  RPCDef in out ps
=   { rpcFunction :: (in,ps) -> (out,ps),
      rpcInitIO   :: InitIO ps  }
::  RPCId in out
```

```
NewRPC  :: (RPCDef in out      *( PState .l .p))
           (l,p)               *(IOState .m .q)
        -> (RPCId  in out,     * IOState .m .q)
ShareRPC:: (RPCDef in out      *( PState .l .p))
            l                  *(IOState .m .p)
        -> (RPCId  in out,     * IOState .m .p)
SendRPC :: (RPCId  in out) in *( PState .l .p)
        -> (SuccessOrFail out,*  PState .l .p)
```

Figure 4.8 The RPC process combinators.

```
NewRPC :: (RPCDef in out *( PState .l .p)) (.l,.p) *(IOState .m .q)
       -> (RPCId  in out,* IOState .m .q)
NewRPC {rpcFunction=f,rpcInitIO=initIO} (l,p) io=:{ioGroups}
=   (rpcId, {io & ioGroups=[group:ioGroups]})
where
    group   = { gPublic    = p,
                gProcesses= [process]}
    process = { ppLocal    = l,
                pInitIO    = [initRPC:initIO],
                pDevices   = []}
    (rpcId, initRPC) = openRPC f
```

Figure 4.9 The definition of `NewRPC`. A list with head elements $a_1 \ldots a_n$ and tail list `a` is denoted by $[a_1, \ldots, a_n : a]$.

The main functions of the definition of `SendRPC` are given in Figure 4.10. The full definition can be found in Appendix B.2-6. For reasons of space we do not repeat the process management data types and operations (these can be found in Chapter 3) and introduce them informally instead: the data type `GState` (*g*lobal *state*) contains the complete process structure of an interactive program. Given the process identification of some interactive process `contextSwitchIn` transforms a given global state into the required process state. Given an arbitrary process state `contextSwitchOut` transforms the process state back into the global state.

The process administration data type is extended with the `RuntimeState` data type (see Appendix B.6). This type reflects the fact that an interactive process is either *running* (the alternative constructor **`Running`**) or *blocked* while waiting for a remote procedure call to be handled (the alternative constructor **`Blocked`**). This alternative is parameterized with the actual remote procedure call request.

Communication with RPC processes is bi-directional and synchronous. Once the connection is established the message is sent and the sender waits for the reply of the RPC process. This reply is sent when the RPC process is ready to reply. As long as this is not the case the sender is *blocked*. In this wait state the system evaluates an arbitrary number of non-blocked interactive processes. This is necessary because the RPC process itself might be blocked waiting for some other interactive process to become unblocked.

How can we model the non-determinism required by the wait loop in a pure functional framework? Recall that every Clean program is a function of type `*World` → `*World` and that this environment is comprised of other environments (Figure 4.1). In our hierarchy of environments we can add a new environment of abstract type `Random` to the `World` environment. The `Random` environment contains a stream of random Boolean values. The only operation available on `Random` is `getRandomBool` :: `Random` → (`Bool`, `Random`) defined by `getRandomBool` [x:`xs`] = (x,`xs`). The types of `openworld` and `closeworld` change to include the `Random` environment. The Random environment is retrieved from the world, together with the Files and Events environments, and placed in the GState process administration by OpenIO. The operation gsGetRandomBool (Appendix B.4) applies getRandomBool to the Random environment stored in the GState record. It should be observed that the condition of the wait loop gsCond waits atleast until the request has been granted and then an arbitrary number of steps. The function switchToSomeProcess which does a context-switch to an arbitrary interactive process of all available processes (and yields its process identification) can use the random stream from the global state analogously.

```
SendRPC :: (RPCId in out) in  *(PState .l .p)
        -> (SuccessOrFail out, *PState .l .p)
SendRPC rpcId in ps=:{procId=id}
|   not (isSuccessful oid)
    =   (Fail,psUnblockThisProcess (contextSwitchIn (id,gs1)))
|   otherwise
    =   (out, psUnblockThisProcess (contextSwitchIn (id,gs3)))
where
    ps1      = psBlockThisProcess rpcId ps
    gs       = contextSwitchOut ps1
    (oid,gs1)= gsLocateRPC rpcId gs
    id1      = getSuccessValue oid
    gs2      = while (gsCond id) gsEval gs1
    (rs,gs3) = gsGetRuntimeState id gs2
    out      = rsGetGrant rs

gsCond :: ProcId GState -> (Bool,GState)
gsCond pId gs
|   not (rsIsGranted rs) = (True, gs1)
|   otherwise            = gsGetRandomBool gs1
where
    (rs,gs1) = gsGetRuntimeState pId gs

gsEval :: GState -> GState
gsEval gs
|   rsIsRunning rs          = contextSwitchOut ps1`
|   rsIsGranted rs          = contextSwitchOut ps1
|   not (isSuccessful oid)  = contextSwitchOut ps2`
|   rsIsBlocked rs` && not (rsIsGranted rs`)
                            = contextSwitchOut ps3
|   otherwise               = contextSwitchOut ps5
where
    (id,ps)  = switchToSomeProcess gs
    (rs,ps1) = psGetRuntimeState ps
    ps1`     = psEval ps1
    req      = rsGetRPCReq rs
    rpcId    = req.rpcRReceiver
    (oid,ps2)= psLocateRPC rpcId ps1
    id1      = getSuccessValue oid
    req1     = {req & rpcROut     = Fail,
                      rpcRGranted = True}
    ps2`     = psSetRuntimeState (Blocked req1) ps2
    (rs`,ps3)= psGetOtherRuntimeState id1 ps2
    (out,ps4)= psCommunicate req id id1 ps3
    req2     = {req & rpcROut     = Success out,
                      rpcRGranted = True}
    ps5      = psSetRuntimeState (Blocked req2) ps4
```

```
psCommunicate :: (RPCReq in out) ProcId ProcId (PState .l .p)
              -> (out, PState .l .p)
psCommunicate req psId pId ps
=   (out, ps4)
where
    gs        = contextSwitchOut ps
    ps1       = contextSwitchIn (pId,gs)
    (f,ps2)   = psGetRPC req.rpcRReceiver ps1
    (out,ps3)= f (req.rpcRIn,ps2)
    gs1       = contextSwitchOut ps3
    ps4       = contextSwitchIn psId gs1
```

Figure 4.10 The definition of `SendRPC`.

The essential component of the definition of `SendRPC` is the wait loop which takes care that an *arbitrary number* of other interactive processes are evaluated before the information exchange is actually dealt with. In comparison with the interleaved event I/O system which applies a deterministic, round-robin interleaving order of interactive processes (see Chapter 3) the system described here is essentially non-deterministic. This has a number of consequences when reasoning about the evaluation of interactive processes. Let *A* be the interactive process that applies `SendRPC` and *B* the RPC process. Then we have the following cases:

(**a**) In case interactive processes of *A*'s group are evaluated or the receiver belongs to *A*'s group then the value of the public process state component can be changed after termination of `SendRPC`.

(**b**) Assume some interactive process, say *C*, does `SendRPC` to *A* (so *A* is an RPC process). *C* detects that *A* is blocked. Consequently *C* blocks and enters the evaluation loop recursively. As soon as *A* has been granted its message from *B*, *C* is allowed to apply the RPC function of *A*, even before the complete receiver function of *A* has been evaluated. So, as in (**a**), the value of the public process state component of *A* may have changed but now also the value of the local process state.

(**c**) Assume some interactive process, say *C*, does `SendRPC` to *B*. If *B* is blocked we have case (**b**) for the situation that *B* is involved in the evaluation of `SendRPC` to yet another RPC process. If *B* is not blocked a number of other processes are evaluated. Either *A* or *C* is granted the reply from *B* first.

Summarising: programmers must be aware that `SendRPC` involves a context-switch. In case of common interactive processes when performing `SendRPC` the result value of the public process state component may differ from the argument value. In case of RPC processes the result value of their local and public process state component value may change.

4.5 Files as global process

In this section the global character of the file system is defined by applying the notion of RPC process introduced in the previous section. Section 4.5.1 describes how RPC processes are applied to localise the file system environment and Section 4.5.2 presents the new definitions of the files operations.

4.5.1 The file server

In this section we present an alternative model of the global character of the file system by introducing one exclusive RPC process, the *file server process* (or *file server*) that handles all file system operations. The local process state component of the file server is the file system environment `Files`, the public process state component is the single constructor type `Nil`. The file system operations of the interleaved event I/O system will be applied only in the file server, albeit with a different name.

The file server is an RPC process of type `FSIn` to `FSOut` (see Figure 4.11). Each `FSIn` and `FSOut` message type alternative holds the arguments of its corresponding `Files` operation (except for the `Files` argument). The RPC function of the file server is `fileServer` (see Figure 4.12). For each **`FSfopenIn`** message `fileServer` replies with an **`FSfopenOut`** message parameterized with the results of `fsfopen` applied to the given arguments and the file server's file system.

```
fsfopen   :: String Int *Files -> ( Bool,*File, *Files)
fssfopen  :: String Int *Files -> ( Bool, File, *Files)
fsfreopen:: *File  Int *Files -> ( Bool,*File, *Files)
fsfclose :: *File      *Files -> ( Bool,       *Files)
fsstdio  ::            *Files -> (*File,       *Files)

::  FSIn  =  FSfopenIn     String Int
          |  FSsfopenIn    String Int
          |  FSfreopenIn   *File  Int
          |  FSfcloseIn    *File
          |  FSstdioIn
::  FSOut =  FSfopenOut    Bool *File
          |  FSsfopenOut   Bool  File
          |  FSfreopenOut  Bool *File
          |  FSfcloseOut   Bool
          |  FSstdioOut         *File
```

Figure 4.11 The renamed `Files` operations of the file server.

```
fileServer :: (FSIn,FSState) -> (FSOut,FSState)
fileServer (FSfopenIn name mode, ps)
=          (FSfopenOut b f,     {ps & pLocal=fs1})
where      (b,f, fs1) = fsfopen name mode ps.pLocal
fileServer (FSsfopenIn name mode,ps)
=          (FSsfopenOut b f,    {ps & pLocal=fs1})
where      (b,f, fs1) = fssfopen name mode ps.pLocal
fileServer (FSfreopenIn f mode,  ps)
=          (FSfreopenOut b f1,  {ps & pLocal=fs1})
where      (b,f1,fs1) = fsfreopen f mode ps.pLocal
fileServer (FSfcloseIn f,        ps)
=          (FSfcloseOut b,      {ps & pLocal=fs1})
where      (b,   fs1) = fsfclose f ps.pLocal
fileServer (FSstdioIn,           ps)
=          (FSstdioOut f,       {ps & pLocal=fs1})
where      (f,   fs1) = fsstdio ps.pLocal
```

Figure 4.12 The file server definition.

4.5.2 Files I/O as remote procedure calls

In this section we show how interactive processes can use the services of the file server. The first major change is that the process state type `PState` looses its `pFiles` field because the file system has now been moved exclusively to the file server. The id of the file server process is stored in the new field `ioFSId` of the `IOState` environment. So we have the following type of process state:

```
:: *PState  l p = { pLocal   :: l,
                    pPublic  :: p,
                    pIOState :: *IOState l p }
:: *IOState l p = { ioFSId   :: RPCId FSIn FSOut,…   }
```

Figure 4.13 The new definition of the process state type.

The files operations are redefined as remote procedure calls with the file server. Figure 4.14 gives the definition of `fopen`, other cases proceed analogously. The type of `fopen` changes because the files environment is not available anymore but should be applied to a process state instead.

```
fopen :: String Int *(PState .l .p) -> (Bool, *File, *PState .l .p)
fopen name mode ps = (b,f,ps1)
where (FSfopenOut b f,ps1)
      = SendRPC ps.pIOState.ioFSId (FSfopenOut name mode) ps
```

Figure 4.14 RPC implementation of files operations.

4.6 Concurrent processes

Does the new semantics of file system operations fit the intuition one expects from concurrent processes? Reconsider the case given in the introduction. Processes *A* and *B* have two process state transition functions: the first, `g`, opens and closes the files f_1 and f_2, and terminates the process (and the other process by sending it a message) if there is a failure. The second, `r`, is a receiver function which on acceptance of some message terminates its interactive process.

```
g :: (PState .l .p) -> PState .l .p
g ps
|     open1 && open2       = ps4
|     otherwise            = seqPIO [QuitIO] ps2`
where (open1, file1, ps1) = fopen  f1 mode ps
      (open2, file2, ps2) = fopen  f2 mode ps1
      (_,            ps3) = fclose file1   ps2
      (_,            ps4) = fclose file2   ps3
      ps2`                 = ASyncSend otherId Nil ps2

r :: x (PState .l .p) -> PState .l .p
r _ ps = seqPIO [QuitIO] ps
```

Figure 4.15 The state transition functions of process *A* and *B*.

One scenario that will cause termination of this program is the following. Choose process *A* for evaluation. The call to `fopen` results in a call to `SendRPC` which blocks *A* until the file server responds. Now *B* is evaluated which also calls `fopen`, which subsequently calls `SendRPC` with the same arguments. So *B* is also blocked until the file server responds. Now we have case (**c**) given in the end of Section 4.4. So either *A* or *B* is granted the result of opening the requested file first. Assume that this occurs for *A*. Then *A* obtains the file f_1. Now *B* is granted the result which contains a False Boolean and an empty file. *B* leaves the wait loop and so does *A*. Whatever the results are for opening the second file, *B* has a False guard and will therefore terminate *A* and *B*.

This program does not terminate only if at every time one process is granted to open f_1 the last granted files operation of the other process was to close f_2. It should be noted that the interleaved semantics is a special case of this scheme.

4.7 Concurrent implementation

The operational semantics of the system as given in Sections 4.4 and 4.5 is Clean code and therefore executable. So we have an implementation that sequentially simulates concurrent processes. In this section we consider how to obtain an implementation of the concurrent event I/O system that allows for concurrent processes using the concurrency primitives of Concurrent Clean (*Concurrent* Clean is the parallel extension to Clean, see Nöcker *et al.* (1992)). With the concurrency primitives of Concurrent Clean a programmer can create new reduction processes that either are evaluated on the current processor (using the `{I}` annotation), or on another processor (using the `{P}` annotation). These reduction processes evaluate the annotated functional expression to root-normal-form. Inter-reduction process communication is demand driven: only if a function requires (part of) the result of the parallel computation then this part is copied to the demanding reduction process.

The inter-process communication as discussed in this chapter is not demand driven and requires a different approach. Section 4.7.1 discusses the concurrent event I/O system without RPC processes and `SendRPC`, and implements the file system operations without these primitives. Section 4.7.2 adds RPC processes and `SendRPC` to the implementation.

4.7.1 Concurrent process groups

Inter-process communication in the concurrent event I/O system without message passing is exclusively through file I/O and data sharing. Data sharing imposes sequentialization of evaluation of interactive processes within one process group due to data dependencies. It is not easy to see how we can create concurrent reduction processes for individual processes, so we focus on process groups instead. Interactive processes from disjoint process groups can interact only via file I/O. Semantically, the computation of a process group is to pair effects to the events of its argument event stream. We have defined this by one loop which for each event accumulates the effects of this event. The same value can be obtained by passing copies of the event stream environment to the process groups which themselves define the event loop for their interactive processes. These results are merged by the system to obtain a new `World` environment value as soon as all process groups have terminated. In this way each process group event loop can be evaluated concurrently.

In a concrete implementation it is not necessary to actually create copies of the event stream nor merge the results because the actual event stream value is never in scope of the program. Instead, we can interface directly with the operating system and access its event queue. An alternative implementation scheme is to create for each new process group one reduction process (using the `{I}` or `{P}` process annotations). The reduction process evaluates the interactive processes of the process group. The reduction process yields a value only at termination. Which value is not important, say an Integer. `OpenIO` checks termination of all process groups simply by matching the values returned by each process group. In this way the termination properties still hold for the implementation. In this scheme it is also not necessary to implement the file system operations by some synchronisation mechanism but instead implement these as operating system calls.

4.7.2 RPC processes and `SendRPC`

The implementation of RPC processes and RPC communication involves three aspects:

(**a**) Creation of RPC processes

(**b**) Implementation of RPC processes

(**c**) Implementation of `SendRPC`

(**a**) RPC processes are special interactive processes that reply to messages of some type `in` with a response of type `out`. We provide the RPC process with a request queue of type `[RPCReq in out]`. Because we implement interactive processes (or rather their process groups) as independent reduction processes, we need to provide a global context to enable reduction processes to locate RPC processes. This can be done by providing the system with a globally accessible table, the *RPC table*, that contains for each RPC process an entry with its `ProcId` and the next free entry to its request queue. Creation of an RPC process (with `NewRPC` or `ShareRPC`) adds a new entry to the RPC table.

(**b**) Given a non-empty request queue the `in out` RPC process takes the first entry and computes the `out` result. It then destructively updates the entry with the `out` result and sets the grant flag to True. The RPC request is removed from the request queue. Like ordinary interactive processes RPC processes request termination by applying `QuitIO`. When applied, `QuitIO` removes the RPC process entry of the process from the RPC table. It can be the case that there still are pending RPC requests of other processes. Therefor each remaining request entry in the request queue is destructively updated with result **`Fail`** and the grant flag is set to True.

(**c**) `SendRPC` first checks in the RPC table whether the RPC process is still present. If not, then the result of the call is **`Fail`**. Otherwise, an RPC request entry with False grant flag is added to the RPC request queue of the RPC process and the interactive process that applied `SendRPC` is blocked. Then the wait loop is entered which evaluates interactive processes from the process group until the request is granted by the RPC process. Observe that this is done by inspection of the same request entry! Once the request is granted the result is redirected to the sending process which is then unblocked.

4.8 Related work

The area of functional operating systems offers some related work with respect to dynamic process creation and inter-process communication. One particular system is the Kent Applicative Operating System project (Turner, 1990). The system is based on earlier work by Stoye (1984). Both systems allow dynamic creation of functional processes. Processes are in essence *stream processors*, functions that transform an ingoing stream to an outgoing stream. Process scheduling is based on evaluation on demand of the outgoing stream and withholding further input. 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.

A different direction is taken in the Facile language (Thomsen *et al.*, 1993). Facile is based on the SML/NJ version (SML, 1993) of Standard ML (Milner *et al.*, 1993) and extends it with higher-order concurrent processes based on CCS (Milner, 1989). It should be noted that Facile is not a pure functional language as it is based on Standard ML. Facile allows the creation of processes defined by arbitrary functions. Inter-process communication is done by means of *channels* which can contain messages of arbitrary type in a type-safe way. One distinct feature of Facile is the ability for separately created processes to communicate using a sort of type library. Type security is checked dynamically, but it is not entirely safe.

4.9 Conclusions

In this chapter we have demonstrated how an interleaved state transition system (the interleaved event I/O system) that is dynamically composed of single state transition systems (interactive processes) can be transformed into a concurrent state transition system (the concurrent event I/O system). We have defined a well-known communication primitive in the framework, the remote procedure call and shown how data dependency by data sharing can be eliminated by application of this primitive. The construction is done entirely on the functional level. This provides us with an executable system which simulates concurrency. We have given some implementation considerations to reach truly concurrently evaluated interactive processes. The concurrent event I/O system can be regarded as an operational semantics of a mini operating system allowing dynamic, interactive, concurrent functional processes.

4.10 Current and future work

As stated in the introduction our final goal are real distributed interactive applications written in a pure functional language. This may require some sophistication of the concurrency model. On the implementation part the granularity of concurrency is now on the level of process groups. We need to consider how to decrease the concurrency granularity to the level of individual interactive processes. Topics related to this problem are making access to shared data into critical sections.