The implementation of interactive local state transition systems in Clean

Peter Achten* and Rinus Plasmeijer

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

Abstract. The functional programming language Clean offers an extensive library, the *object I/O library*, to programmers to create interactive applications that use Graphical User Interfaces (GUI). Programs are constructed by composition of interactive objects. Interactive objects share state. To support modular programming it is also possible to provide objects with local state. Hierarchies of stateful objects are defined by means of algebraic specifications. Good use is made of the fact that Clean is a strongly typed language. In this paper we show how the evaluation of interactive objects is implemented in the object I/O library. The evaluation can be handled elegantly using lazy evaluation.

1 Introduction

For many years the functional programming language Clean [7, 13, 14] enables programmers to create efficient, interactive applications that use Graphical User Interfaces (GUIs). This is done by means of an extensive library that has been developed in Clean. The first publicly available version was the 0.8 I/O library [1, 2]. Newer versions have emerged, culminating in the current publicly available *object I/O library* [3, 4], version 1.1. Although the object I/O library is very different from the very first system, the basic concepts have remained the same. Briefly, these are:

- Pure functional I/O is realised using the *uniqueness type system* [16, 5] of Clean. The uniqueness type system allows programmers to manipulate values that represent parts of the outside world in a safe, flexible, and efficient way. Such kind of values are called *environments*.
- Interactive programs are *state transition systems*. The programmer defines an initial 'logical' state of arbitrary type. GUI objects are defined on a high level of abstraction using algebraic types. The behaviour of GUI components are defined by higher-order function arguments. These functions are the state transitions. When needed, these functions will be applied to the proper actual value of the state. This is taken care of completely by the run-time system of the I/O library.

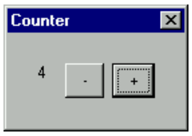
* Supported by STW under grant NIF33.3059: The Concurrent Clean System.

– Given the initial state and initial GUI components, the interactive program can be handled completely by the I/O library. All OS resource management and low-level event handling is done internally.

The main research theme that pushed the development of the object I/O library was the question how to obtain a modular style of program construction given the basic concepts. This has lead to the incorporation of *interactive processes* and *local state*. The design decisions have already been introduced elsewhere [3]. In this paper we will focus on the concept of local state and how it has been implemented in the object I/O system.

Let us illustrate our goal by means of a simple example, that of a reusable *counter* component. Below is the definition, the figure shows the counter component when created in a dialogue.

```
counter id i
= CompoundControl
  {newLS =i
  ,newDef=TextControl (toString i) [ControlId id]
            :+:
          ButtonControl "-" [ControlFunction (set (-1))]
            :+:
          ButtonControl "+" [ControlFunction (set 1)]
  } []
```



The counter component consists of a text control that will display the current count, and two button controls, labelled `-` and `+`, to allow the user to decrement and increment the current count. These controls are placed in a compound control which introduces a local layout scope. The important features of this specification are:

– The controls *share* a local integer state which is *hidden* from any context in which the control will be placed. The type of the local state is polymorphic. We ensure that objects that share local state are parameterised with the correct type. Hiding is obtained using *existential quantification* of the local state type. This application of the type system gives us fine grained control on local state.
– The function `counter`, given an identification value `id` and an initial count value `i` returns a *data structure*. No GUI resources are created at this point. Using data types for GUI specifications turns them into first class citizens: we can write functions that manipulate GUI specifications using standard functional techniques.
– The behaviour of the button controls is given by the state transition function `set`, which is parameterised with the value by which the local counter value must change. State transition functions are always applied to a pair of its local and global state. The global state is required to be able to manipulate the world: files can be changed, GUI components can be created and closed, and so on. In this case `set` changes the local count value and provides visual feedback to the user by changing the text field of the text control:

```
set dx (curcount,state)
     = (newcount,setControlText id (toString newcount) state)
where
     newcount = curcount+dx
```

It should be noted that `set` is applied to the *current* local and global state. The result value of `set` is the *new* local and global state. Subsequent state transition functions will be applied to the new state values.

This approach improves on the 0.8 I/O library in which there is only one global program state. This requires the programmer to mimick local state in the program state. Although possible, it forces programmers to code local state management, which leads to less clear and error prone programs. In the approach as sketched above all state is managed by the runtime system of the object I/O library.

The example of the counter illustrates that interactive object I/O programs are specifications of *local state transition systems* that are evaluated by the object I/O library. Assume that the program creates the dialog as shown above:

```
openDialog dialogstate (Dialog "Dialog" (counter id 0) []) globalstate
```

What happens is that the `openDialog` function takes the dialog definition `(Dialog "Dialog" (counter id 0) [])` and creates the proper OS resources. These are stored in the global state, together with the local state values. In this case, these are the local dialog state `dialogstate` and the local integer state of `counter`. As a result, the dialog appears on screen and can be manipulated by the user.

Now assume that the user selects the `+` button. This generates some OS event containing an internal handle identifying the button at OS level. The evaluation function of the object I/O library will derive that the button has been pressed and that, as a consequence, the callback function `(set id 1)` must be evaluated. It collects the local state component values required by `set` and also the current global state value. The callback function increments the local state 0 to 1 and uses the global state to change the text value of the text control. After evaluation, the object I/O library stores the new local state component values and global state component values, so that these will be passed to subsequent state transitions.

This example illustrates that besides managing OS resources for GUI objects, one major task of the evaluation mechanism of the object I/O library is to evaluate the local state transitions. The evaluation mechanism can be written as a pure function. The remainder of this paper is dedicated to its implementation. The key idea of the algorithm is that while constructing the local state argument of a callback function, the evaluation function also adds forward references to the result local state components in the global state. This can be done by virtue of *lazy evaluation*. Although the algorithm was developed for the object I/O system, it can be generalised to non-GUI application domains. In Section 2 we show how this can be done. It allows us to focus on the algorithm without getting distracted by the many details required when implementing this technique in a useful GUI library. Its incorporation in the object I/O library is presented in Section 3. In Section 4 we analyze the performance of the systems presented in this paper. Related work is discussed in Section 5 and we conclude in Section 6.

2 Generalised state transition systems

In this section we introduce generalised state transition systems. We start with very simple *global* state transition systems in Section 2.1, and continue with *local* state transition systems in Section 2.2. The library developer is responsible for two things: the construction of the set of types that are used by application programmers to specify state transition systems, and the state transition evaluation function that computes the state transitions given such a specification.

2.1 Global state transition systems

Global state transition systems consist of a 'logical' *program state* of arbitrary type and a structured set of functions that change the logical state and the set of state transition functions. It is our goal to capture this concept by a collection of polymorphic types. Given an arbitrary type `ps` for the program state, then we, as the library developer, construct a type `(T ps)` that contains the state transition functions. These functions have type `(IdFun (ps,T ps))`. (For conciseness, we use the synonym type `IdFun` defined as `IdFun x :== x->x`.) The type constructor `(T ps)` is either an object (that can be identified by means of an integer value) with a state transition function, or a list of elements of type `(T ps)`. If we also introduce the synonym type `(GSTS ps)` for the pair of program state of type `ps` and state transition system of type `(T ps)`, then we have the following simple set of types:

```
::  GSTS ps :== (ps,T ps)
::  T    ps =   Obj  Int (IdFun (GSTS ps))
            |   List [T ps]
```

An application programmer can specify a global state transition system by defining a value of type `(GSTS ps)`. This is the *initial state* of the global state transition system. The state transitions are calculated by a function that is predefined by the library developer and hidden from the application programmer. This function, `nextstate`, selects a state transition function and applies it to the state transition system. Assume that we have a function `getobj :: Int (GSTS ps) -> (Bool,IdFun (T ps),GSTS ps)` that finds the indicated state transition function if it exists. Now the definition of `nextstate` is easy:

```
nextstate :: Int (GSTS ps) -> (GSTS ps)
nextstate i (ps,t)
    # (found,f,t) = getobj i t
    | not found   = (ps,t)
    | otherwise   = f (ps,t)
```

The type definition of `nextstate` might be confusing. In Clean functions have *arity* (2 in case of `nextstate`). Function arguments in the type definition are separated by whitespace. The result type is preceded by `->`.

2.2 Local state transition systems

Local state transition systems extend global state transition systems with local states of arbitrary type. Such a system allows the programmer to introduce new local state values at any place in the recursive data structure and define local state transition functions that change both the local state and global state. When introducing a local state we want to specify which objects are able to modify that particular local state. These objects *share* the local state. The local state must be *encapsulated* to prevent access by other objects.

It is our task to construct a set of types that can be used by application programmers to define local state transition systems. Analogous to global state transition systems, we define a type `(LSTS ps)` that contains the global, program state of type `ps` and the recursive type `(T ps)` that contains the collection of objects that have local state and local state transition functions.

```
::  LSTS ps :== (ps,T ps)
::  T    ps :== [Obj ps]
```

A top-level object of type `(Obj ps)` introduces a local state of some type `ls`. The top-level object consists of component objects that modify the local state of type `ls` and the local state transition system. Their type definition is therefore `(Comp ls (LSTS ps))`. The type of the local state is encapsulated using *existential quantification*. In Clean, a type constructor can introduce existentially quantified type variables on the right-hand side of the type constructor by defining them between `E.` and `:`.

```
::  Obj ps = E. ls: Obj ls [Comp ls (LSTS ps)]
```

A component object that modifies a local state of type `ls` and a local state transition system of type `lsts` can be one of three things: *(a)* it contains a local state transition function of type `(IdFun (ls,lsts))`, or *(b)* it encapsulates a private local state of some type `new` and ignores the local state `ls` of its context, or *(c)* it encapsulates a private local state of some type `new` and also changes the local state `ls` of its context. Objects *always* have access to the global state `lsts`. This is defined with the following type `Comp` which has three alternatives for the cases described above.

```
::  Comp  ls lsts = Comp Int (IdFun (ls,lsts)) // (a)
                  | NewLS (NewLS ls lsts)      // (b)
                  | AddLS (AddLS ls lsts)      // (c)
::  NewLS ls lsts = E. new:{ newLS::new, newDef::[Comp  new     lsts] }
::  AddLS ls lsts = E. new:{ addLS::new, addDef::[Comp (new,ls) lsts] }
```

Given these type definitions the application programmer can construct hierarchical structures of objects that change a global state (of type `(LSTS ps)`) and that can introduce at arbitrary depth new local state values. These can be *shared* by arbitrary collections of objects, but only those that have it in scope. The type system ensures that the local state values are properly encapsulated and that the objects contain local state transition functions of the correct type.

The final step is to provide an implementation of `nextstate`. In contrast to global state transition systems, `nextstate` must now construct the proper local state argument of each local state transition function. After applying the local state transition function, the new local state components must be restored in the recursive data structure. We show that this can be solved elegantly by making handy use of the lazy evaluation mechanism of Clean. The crucial point is that while `nextstate` is collecting the local state components, it immediately adds (lazy) references to the result local state values in the recursive data structure.

We assume that we have a function that finds the `(Obj ps)` value identified by a given integer `getobj :: Int (LSTS ps) -> (Bool,Obj ps,LSTS ps)` and a function that replaces an `(Obj ps)` value identified by a given integer `setobj :: Int (Obj ps) (T ps) -> (T ps)` in the recursive data structure `(T ps)`. These functions can be defined easily.

Now `nextstate` can be defined as follows: first retrieve the identified object using `getobj`, then calculate the new object state and state transition system using `nextobjstate`, and finally return the new state transition system by replacing the old object with the new object. It should be noted that, as promised, the state transition argument of `nextobjstate` contains a reference to the result of `nextobjstate` itself, namely `obj1`.

```
nextstate :: Int (LSTS ps) -> (LSTS ps)
nextstate i (ps,t)
    # (found,obj,t)     = getobj i t
    | not found         = (ps, t )
    | otherwise         = (ps1,t2)
    with
        (obj1,(ps1,t2))  = nextobjstate i obj (ps,setobj i obj1 t)
```

The `(Obj ps)` argument of `nextobjstate` contains all the local state components required to evaluate the proper local state transition function. The first local state component is contained in the `(Obj ps)` argument itself. The function `nextobjsstate` recursively applies the function `nextobjstate'` given below to each of the component objects in the list argument of `(Obj ps)` until a state transition function has been applied. The definition of `nextobjsstate` is straightforward and we omit it for brevity.

```
nextobjstate :: Int (Obj ps) (LSTS ps) -> (Obj ps,LSTS ps)
nextobjstate i (Obj ls objs) lsts
    # (_,objs,(ls,lsts)) = nextobjsstate i objs (ls,lsts)
    = (Obj ls objs,lsts)
```

More interesting are the alternatives of `nextobjstate‘`. For each case *(a)*, *(b)*, and *(c)* as introduced above, it has one alternative that handles the appropriate local state component construction. In *(a)* the local state transition function is applied if the object is actually found:

```
nextobjstate‘:: Int (Comp ls lsts) (ls,lsts)
            -> (Bool,Comp ls lsts, (ls,lsts))
nextobjstate‘ i (Comp j f) (ls,lsts)
    | i==j      = (True, Comp j f,f (ls,lsts))
    | otherwise = (False,Comp j f,  (ls,lsts))
```

In *(b)* the new local state value is passed to the component objects while the context local state component is ignored. After evaluation, the new local state value and component objects are placed in the new object structure:

```
nextobjstate‘ i (NewLS {newLS=new,newDef=objs}) (ls,lsts)
    # (done,objs,(new,lsts)) = nextobjsstate i objs (new,lsts)
    = (done,NewLS {newLS=new,newDef=objs},(ls,lsts))
```

In *(c)* the context local state is extended with the new local state value and passed on to the component objects. After evaluation, the new extended local state value and component objects are placed in the new object structure:

```
nextobjstate‘ i (AddLS {addLS=new,addDef=objs}) (ls,lsts)
    # (done,objs,((new,ls),lsts)) = nextobjsstate i objs ((new,ls),lsts)
    = (done,AddLS {addLS=new,addDef=objs},(ls,lsts))
```

It should be observed that the definitions of the basic functions `getobj`, `setobj`, and `nextobjstate‘` straightforwardly follow the recursive type definitions of `(T ps)`. The only piece of ‘magic’ occurs at the definition of `nextstate` where a forward reference is constructed by virtue of lazy evaluation.

3 The object I/O state transition system

In the previous section we have shown how to construct a set of type definitions that allow application programmers to define general local state transition systems. Such a specification is then evaluated by a hidden predefined library function that calculates the proper local state transitions. In this section we show how this approach is applied in the object I/O system.

We first explore the set of type definitions that we have constructed for application programmers to construct GUI components (Section 3.1). The object I/O library maps GUI component definitions to an internal representation and creates the corresponding OS resources (Section 3.2). This will cause the GUI components to appear on screen and they can subsequently be modified by the user of the program. These user actions generate OS events. Each user action triggers the evaluation of the proper local state transition function (Section 3.3).

3.1 Types as a specification language

The set of type definitions provided for application programmers to create interactive object I/O programs has a very similar top-level structure as the type definitions presented in Section 2.2. The type `(LSTS ps)` corresponds with the *process state* type `(PSt ps)`. Instead of a tuple type, the process state is a record type, defined as `*PSt ps = {ps::ps, io::*IOSt ps}`. (In the actual object I/O library, `PSt` has an additional type variable, but we omit this for conciseness.) The `ps` component is the *program state*. It contains the data that is needed during evaluation of the interactive program and which is globally accessible to all GUI components. The type `(T ps)` corresponds with the abstract type `(IOSt ps)` (*I/O state*), and can be found in the `io` record field of `(PSt ps)`. It is an abstract value because it contains all OS resources of the GUI components that are created and manipulated by the state transition functions.

The I/O state is abstract. A programmer can add and remove GUI objects by appropriate library functions at any time during evaluation of the interactive program. The I/O state is initialised by an initialisation function. This is simply the very first state transition function of the interactive program, and is of type `(IdFun (PSt ps))`. There are GUI creation functions for menus, windows, dialogues, timers, and receivers. In this paper we will restrict ourselves to dialogues and only two of their component objects, button and text controls.

GUI components are defined by *algebraic types*. We use the following convention: the type constructor has as the final two type variable arguments `ls` and `pst`. The first of these will always correspond to the local state, and the second with the process state. The type constructor has a data constructor with the same name. (This is possible due to separate name spaces.) The number of mandatory component attributes is kept as small as possible. There is always a list of optional attributes. When a GUI component is composite (dialogue), then its component objects are represented by means of a *type constructor variable*. The only restriction imposed on the type constructor variable is that it is defined on the same local state `ls` and process state `pst` types. Here is the set of type definitions for dialogues, button and text controls:

```
::  Dialog         c ls pst
 =  Dialog String (c ls pst) [WindowAttribute  (ls,pst)]
::  ButtonControl    ls pst
 =  ButtonControl String     [ControlAttribute (ls,pst)]
::  TextControl      ls pst
 =  TextControl   String     [ControlAttribute (ls,pst)]
```

The optional attributes are parameterised with the pair of the local state and process state. The only attributes that we will be interested in in this paper are the *identification* attribute (`WindowId` and `ControlId`) and the local state transition function attribute (`WindowFunction` and `ControlFunction`). Here are the type definitions of window attributes (left) and control attributes (right):

```
::  WindowAttribute st                 ::  ControlAttribute st
```

```
 =  WindowId         Id                     =  ControlId       Id
 |  WindowFunction   (IdFun st)             |  ControlFunction (IdFun st)
 |  ...                                     |  ...
```

Interactive object I/O programs have only indirect access to their GUI components. The program has to identify the objects that are required to be modified or accessed during run-time. The identification value is of abstract type `Id`. Values are created by appropriate object I/O library functions. Every `Id` value is bound at most once per open GUI object to allow unambiguous identification.

The rules that define which components are allowed to be glued are given by means of *type constructor class* instances. Using type constructor classes allows one to add new GUI component types or define your own type definitions without changing the existing library. The type constructor class `Dialogs` has a member function `openDialog` which expects an initial local state value and dialogue definition. A proper instance of a dialogue is the type constructor `(Dialog c)`, provided that the type variable `c` is a proper instance of the `Controls` type constructor class. The latter class defines which type constructors can be used as control elements of a dialogue. Its member function `toControlSt` maps every `Controls` instance type to an internal, abstract representation of type `ControlSt` (Section 3.2). `ButtonControl` and `TextControl` are two of its instances.

```
class Dialogs ddef where
    openDialog :: .ls !(ddef .ls (PSt .ps)) !(PSt .ps) -> (PSt .ps)
    ...
class Controls cdef where
    toControlSt :: !(cdef .ls (PSt .ps)) !(PSt .ps)
          -> (![ControlSt .ls (PSt .ps)],!(PSt .ps))
    ...

instance Dialogs  (Dialog c) | Controls c
instance Controls ButtonControl
instance Controls TextControl
```

In the object I/O library a number of type constructor combinators are defined to allow the easy composition of objects that work on the same local state and global state. These type constructor combinators are also proper instances of the `Controls` type constructor class:

```
::  :+:    c1 c2 ls pst = (:+:) infixr 9 (c1 ls pst) (c2 ls pst)
::  ListLS    c  ls pst = ListLS [c ls pst]
::  NilLS        ls pst = NilLS

instance Controls ((:+:) c1 c2) | Controls c1 & Controls c2
instance Controls (ListLS c)    | Controls c
instance Controls NilLS
```

Here are some examples of their use. Control `c1` glues a button control and a text control, `c2` creates a list of button controls using a list comprehension, and `c3` defines no controls at all. The images are screen shots of the same controls when created in a dialogue with definition `(Dialog "Dialog" c [])`, with `c` equal to `c1`, `c2`, or `c3` respectively.

```
c1 = ButtonControl "Hello" [] :+: TextControl "world!" []
c2 = ListLS [ButtonControl (toString i) [] \\ i<-[1..3]]
c3 = NilLS
```

In Section 2.2 we have used two types `NewLS` and `AddLS` to introduce encapsulated local state. These types are also predefined in the object I/O library in a more general form, using type constructor variables:

```
::  NewLS c ls pst = E. new: {newLS::new, newDef:: c  new     pst}
::  AddLS c ls pst = E. new: {addLS::new, addDef:: c (new,ls) pst}

instance Controls (NewLS c) | Controls c
instance Controls (AddLS c) | Controls c
```

The purpose of `NewLS` and `AddLS` is the same as in Section 2.2. `NewLS` allows a control to ignore the context local state, and instead introduce a new local state, while `AddLS` allows its component objects to change both the context local state *and* change a new local state component. Here are two examples:

```
c1={ newLS =0
   , newDef=ButtonControl "count" [ControlFunction (\(i,pst)->(i+1,pst))]
   }
c2={ addLS =0
   , addDef=ButtonControl "count" [ControlFunction (\((i,_),pst)
                                        ->((i+1,isEven (i+1)),pst))]
   }
```

The first control, `c1`, defines a button control that increments a local integer state that has initial value zero. This control can be glued in combination with any other control. This does not hold for `c2`: it defines a button control that expects a local state context of type `Bool`. It uses the boolean local state component to report whether its internal integer local state component is even (using the library function `isEven`).

3.2 Creation of GUI objects

It is the task of the programmer to define the appropriate GUI components that have to be used in the application. Given such a GUI definition, which is a data structure, it is then the task of the object I/O library to create the corresponding

OS resources and manage the concrete GUI components as required on the current platform. These are stored as an internal data structure in the `IOSt` component of the process state. This data structure must be able to accomodate all local state components, all local state transition functions, and the required OS resources. The mapping is done by the member functions of the corresponding type constructor classes.

The `Dialogs` type constructor class member function `openDialog` maps a local state value of type `ls` and an expression of type `(Dialog c ls (PSt ps))` (provided that `c` is an instance of the `Controls` type constructor class) to a `(DialogSt (PSt ps))` (for simplicity we leave out the OS stuff):

```
::  DialogSt pst = E. ls:{ dialogLS::ls, controls::[ControlSt ls pst] }

instance Dialogs (Dialog c) | Controls c where
    openDialog ls (Dialog title controls atts) pst
        # (controlSts,pst) = toControlSt controls pst
        # dialogSt         = { dialogLS=ls, controls=controlSts}
        = {pst & io=setDialogSt dialogSt pst.io}
```

The controls of a dialog are mapped to `ControlSt` values using the `Controls` class member function `toControlSt`. `ControlSt` is an algebraic data type. Its definition is almost identical to the `Comp` type definition in Section 2.2:

```
::  ControlSt    ls pst = ControlSt Id  (IdFun (ls,pst))
                        | ControlListLS [ControlSt    ls pst]
                        | ControlNewLS  (ControlNewLS ls pst)
                        | ControlAddLS  (ControlAddLS ls pst)
::  ControlNewLS ls pst
 =  E. new:{ newLS::new, newCS::[ControlSt  new     pst] }
::  ControlAddLS ls pst
 =  E. new:{ addLS::new, addCS::[ControlSt (new,ls) pst] }
```

Standard controls, such as buttons and text controls are mapped to the alternative `ControlSt`. For simplicity, we assume that the identification value and attributes are the only things that are stored. In the actual implementation this is the place where metrics are calculated and actual OS resources are allocated.

```
instance Controls ButtonControl where
   toControlSt (ButtonControl title [ControlId id,ControlFunction f]) pst
       = ([ControlSt id f],pst)
instance Controls TextControl where
   toControlSt (TextControl title [ControlId id,ControlFunction f]) pst
       = ([ControlSt id f],pst)
```

The type constructor combinators `:+:`, `ListLS`, and `NilLS` are mapped to the `ControlListLS` alternative:

```
instance Controls ((:+:) c1 c2) | Controls c1 & Controls c2 where
   toControlSt (c1 :+: c2) pst
```

```
     # (controlSts1,pst) = toControlSt c1 pst
     # (controlSts2,pst) = toControlSt c2 pst
     = ([ControlListLS (controlSts1 ++ controlSts2),pst)
instance Controls (ListLS c) | Controls c where
   toControlSt (ListLS cs) pst
     # (controlSts,pst) = seqList (map toControlSt cs) pst
     = ([ControlListLS (flatten controlSts)],pst)
instance Controls NilLS where
   toControlSt NilLS pst
     = ([],pst)
```

The local state type constructor combinators `NewLS` and `AddLS` are mapped to `ControlNewLS` and `ControlAddLS` respectively:

```
instance Controls (NewLS c) | Controls c where
   toControlSt {newLS=new,newDef=c} pst
     # (controlSts,pst) = toControlSt c pst
     = ([ControlNewLS {newLS=new,newCS=controlSts}],pst)
instance Controls (AddLS c) | Controls c where
   toControlSt {addLS=new,addDef=c} pst
     # (controlSts,pst) = toControlSt c pst
     = ([ControlAddLS {addLS=new,addCS=controlSts}],pst)
```

As one can see, the structure of these mappings is straightforward. What we end up with after opening a dialogue is that the process state `(PSt ps)` result of `openDialog` contains everything the object I/O system needs to handle the dialogue and its controls. This is handled in the next section.

3.3 Handling OS events: calculating state transitions

We can now discuss the implementation of the state transition function `nextstate`. It uses exactly the same technique as described in Section 2.2. The functions `getDialogSt` and `setDialogSt` retrieve and restore respectively the `DialogSt` from the `IOSt`. They correspond to the `getobj` and `setobj` functions in Section 2.2. The function `nextdialogstate` corresponds with `nextobjstate`.

```
nextstate :: Id (PSt .ps) -> PSt .ps
nextstate id pst=:{io}
    # (found,dialogSt,io)= getDialogSt id io
    | not found          = {pst & io=io}
    | otherwise          = pst1
    with
        (dialogSt1,pst1) = nextdialogstate id dialogSt
                                {pst & io=setDialogSt dialogSt1 io}
```

The definition of the function `nextdialogstate` and its component functions are completely analogous to the functions as described in Section 2.2 (`nextobjstate` and its component functions). We therefore include their definitions without further comment. (We omit the definition of `nextcontrolsstate` for conciseness. It recursively applies the `nextcontrolstate` function to each control state in the argument list until it has been found.)

```
nextdialogstate :: !Id !(DialogSt (PSt .ps)) !(PSt .ps)
                    -> (!DialogSt (PSt .ps),   PSt .ps)
nextdialogstate id dialogSt=:{dialogLS=ls,controls=cs} pst
    # (_,cs,(ls,pst)) = nextcontrolsstate id cs (ls,pst)
    = ({dialogSt & dialogLS=ls,controls=cs},pst)

nextcontrolstate :: !Id !(ControlSt .ls .pst) *(.ls,.pst)
               -> (!Bool,!ControlSt .ls .pst, *(.ls,.pst))
nextcontrolstate id c=:(ControlSt controlId f) ls_pst
    | id==controlId = (True, c,f ls_pst)
    | otherwise     = (False,c,  ls_pst)
nextcontrolstate id (ControlListLS cs) ls_pst
    # (done,cs,ls_pst) = nextcontrolsstate id cs ls_pst
    = (done,ControlList cs,ls_pst)
nextcontrolstate id (ControlNewLS {newLS=new,newCS}) (ls,pst)
    # (done,cs,(new,pst)) = nextcontrolsstate id newCS (new,pst)
    = (done,ControlNewLS {newLS=new,newCS=cs},(ls,pst))
nextcontrolstate id (ControlAddLS {addLS=new,addCS}) (ls,pst)
    # (done,cs,((new,ls),pst))= nextcontrolsstate id addCS ((new,ls),pst)
    = (done,ControlAddLS {addLS=new,addCS=cs},(ls,pst))
```

4 Performance

In the sections 2.1 and 2.2 we have introduced global and local state transition systems. Global state transition systems form the basis of the 0.8 I/O library, while local state transition systems form the basis of the object I/O library. Local state transition systems offer application programmers more expressive power by allowing the introduction of polymorphic local state. One cannot expect to get this for free. In this section we investigate the performance penalty one has to pay for the additional expressiveness.

When we compare the runtime performances of global and local state transition systems as defined in sections 2.1 and 2.2 we can see that there is indeed a performance penalty. The execution times of local state transition systems are approximately a factor of two longer than equivalent global systems.

A similar factor should also show up when comparing equivalent programs written for the 0.8 I/O library and the object I/O library. However, this is not the case. To compare the run-time efficiency we have written a program in both libraries that counts the number of state transitions per second. In this way we can derive the run-time overhead of the libraries. To do this so-called *zero timers* are used. A zero-timer attempts to evaluate its state transition function as often as possible (but still triggered by events). It turned out that the current implementation of the object I/O system is actually 30% faster (probably due to better implementation of timers). The number of state transitions per second achieved were 1500 by the 0.8 I/O program and 2200 by the object I/O 1.2

program. This is sufficient for any practical event driven application. At first sight one might argue that a GUI program never requires more than something in the order of ten state transitions per second. This is not the case. Consider for instance scrolling through a piece of text, mouse tracking, and using timers for time-consuming user-interruptable actions such as rendering fractals, ray-tracing, or 3D images. In these cases the run-time overhead should be minimal.

5 Related work

In the context of pure functional languages a great diversity of solutions to their integration with GUIs have emerged. The approaches can be roughly characterised as (event) stream style and monadic style. Examples of stream based systems are FUDGETS [8] and *gadgets* [12]. In both systems local state must be provided by application programmers via recursive stream handling functions. State information can only be shared via stream communication. In [17] type constructor classes have been used in an elegant way to specify the relationship between configuration options (attributes in our terminology) and widgets (GUI objects). In this approach local state in a widget can be mimicked by applications by setting a new callback function to that widget that is partially parameterised with the local state value. The I/O system in Opal [9] relies on a monadic non-deterministic choice operator and concurrent evaluation of agents. In the GUI framework state can be incorporated in a gate using a recursive transformation function. This state is globally accessible. The *pidgets* system [15] offers a high-level interface to PostScript like graphics and a tight integration with widgets and mouse handling. State is incorporated with global variables based on mutable variables. The system supports unrestricted concurrency.

The object I/O system differs from these system by its explicit relationship between GUI objects and (local) state. The programmer only defines initial state values and can focus on the state transition functions. Another difference is that we want to define GUI objects by means of data structures. This allows one to define functions that pass around GUI specifications that can be manipulated and changed before they are actually created.

Although not discussed in this paper, the state transition model supports interactive processes. An interactive proces is just another state transition system that can be evaluated in an interleaved, sequential way (for details, see [3]). To obtain a concurrent semantics, one only needs to include a concept of *context switching* when a state transition function is changing the environment. The resulting system can still be evaluated sequentially. This contrasts strongly with the general opinion that one has to use a concurrent functional language to construct GUI programs flexibly [15, 9].

6 Conclusions and future work

In this paper we have presented the basic evaluation mechanism of the Clean object I/O system. This mechanism is written completely in Clean and is capable of handling arbitrary collections of GUI components that share and encapsulate local polymorphic state. An elegant implementation has been achieved due to the lazy evaluation strategy of Clean. We have shown that the same technique can also be applied in a more generalised version, for any strongly typed and lazy functional programming language.

To judge the performance of the implementation and quality of the design of the object I/O library we are using it in real-world applications. The currently available Clean distribution has an Integrated Development Environment (IDE), which is written in the 0.8 I/O library. We have been redesigning and extending the IDE using the object I/O library. This is an excellent test case because the implementation of an IDE uses virtually every aspect of a GUI library. Inefficiencies also immediately show up, as users expect a development environment to respond promptly. The working version 1.2 of the object I/O library incorporates many improvements over the 1.1 version due to this experience. The resulting application is used in daily practice at our department for all Clean programs developments on Microsoft Windows PCs (as this is the only currently supported platform for version 1.2).

Our current main effort is to release the 1.2 version of the object I/O library for the Microsoft Windows platform. This will be followed by a Macintosh version. Improving efficiency is a never ending story. As mentioned in the previous section, the object I/O system has a concurrent semantics, although the implementation is sequential. We intend to exploit the concurrency facilities of Concurrent Clean to investigate how to improve the implementation. Longer term future work is to make the object I/O library suitable for heterogenous networks to create distributed interactive applications.

Acknowledgements

The implementation of the object I/O library has been improved significantly due to the feedback of the users, in particular Diederik van Arkel who is constructing the new Clean IDE. The authors would also like to thank the anonymous referees for their constructive criticism.

References

[1] Achten, P.M., van Groningen, J.H.G., and Plasmeijer, M.J. High Level Specification of I/O in Functional Languages. In Launchbury, J., Sansom, P. eds., *Proceedings Glasgow Workshop on Functional Programming*, Ayr, Scotland, 6-8 July 1992. Workshops in Computing, Springer-Verlag, Berlin, 1993, pp. 1-17.

[2] Achten, P.M. and Plasmeijer, M.J. The ins and outs of Clean I/O. In *Journal of Functional Programming* **5**(1) - January 1995, Cambridge University Press, pp. 81-110.

[3] Achten, P.M. and Plasmeijer, M.J. Interactive Functional Objects in Clean. In Clack, C., Hammond, K. and Davie, T. eds., *Proceedings 9th International Workshop Implementation of Functional Languages, IFL'97*, St. Andrews, Scotland, UK, September 1997, selected papers, LNCS **1467**, Springer, pp. 304-321.

[4] Achten, P.M. and Wierich, M. A Tutorial to the Clean Object I/O Library - version 1.1. *Internal Report*, Department of Functional Programming, University of Nijmegen, The Netherlands, 1999. Available on Internet:
`ftp://ftp.cs.kun.nl/pub/Clean/supported/ObjectIO/doc/tutorial.11.ps.gz`.

[5] Barendsen, E. and Smetsers, J.E.W. Uniqueness Type Inference. In [10], pp. 189-206.

[6] Bjørner, D., Broy, M., Pottosin, I.V. eds. *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference Akademgorodok*, Novosibirsk, Russia, June 25-28, 1996, Proceedings, LNCS **1181**, Springer.

[7] Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, and Plasmeijer, M.J. Clean: A Language for Functional Graph Rewriting. In Kahn. G. ed. *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.

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

[9] Frauenstein, Th., Grieskamp, W., Pepper, P., and Südholt. Communicating Functional Agents and Their Application to Graphical User Interfaces. In [6], pp. 386-397.

[10] Hermenegildo, M. and Swierstra, S.D. eds. *Proceedings of Seventh International Symposium on Programming Languages: Implementations, Logics and Programs*, Utrecht, The Netherlands, 20-22 September 1995, LNCS **982**, Springer-Verlag.

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

[12] Noble, R. and Runciman, C. Gadgets: Lazy Functional Components for Graphical User Interfaces. In [10], pp. 321-340.

[13] Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. Concurrent Clean. In Aarts, E.H.L., Leeuwen, J. van, Rem, M., eds., *Proceedings of Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202-219.

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

[15] Scholz, E. PIDGETS - Unifying Pictures and Widgets in a Constraint-Based Framework for Concurrent GUI Programming. In Kuchen, H., Swierstra, S.D. eds. *Proceedings of eighth International Symposium on Programming Languages: Implementations, Logics, and Programs*, Aachen, Germany, September 1996, LNCS **1140**, Springer, pp. 363-377.

[16] Smetsers, J.E.W., Barendsen, E., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In Schneider, H.J., Ehrig, H. eds. *Proceedings Workshop Graph Transformations in Computer Science*, Dagstuhl Castle, Germany, January 4-8, 1993, LNCS **776**, Springer-Verlag, Berlin, pp. 358-379.

[17] Vullinghs, T., Schulte, W., and Schwinn, Th. The Design of a Functional GUI Library Using Constructor Classes. In [6], pp. 398-408.