High Level Specification of I/O in Functional Languages

Peter Achten, John van Groningen, Rinus Plasmeijer
revised version, November 1992
University of Nijmegen, The Netherlands
peter88@cs.kun.nl, john@cs.kun.nl, rinus@cs.kun.nl

Abstract

The interface with the outside world has always been one of the weakest points of functional languages. It is not easy to incorporate I/O without being allowed to do side-effects. Furthermore, functional languages allow redexes to be evaluated in any order while I/O generally has to be performed in a very specific order. In this paper we present a new solution for the I/O problem which we have incorporated in the language Concurrent Clean. Concurrent Clean offers a linear type system called *Unique Types*. It makes it possible to define functions with side-effects *without violating the functional semantics*. Now it is possible to change any object in the world in the way we wanted: e.g. arrays can be updated in-situ, arbitrary file manipulation is possible. We have used this powerful tool among others to create a library for window based I/O. Using an explicit environment passing scheme provides a high-level and elegant functional specification method for I/O, called Event I/O. Now the specification of I/O has become one of the strengths of functional languages: interactive programs written in Concurrent Clean are concise, easy to write and comprehend as well as efficient. The presented solution can in principle be applied for any other functional language as well provided that it actually uses graph rewriting semantics in the implementation.

1 Introduction

A lot has been said in favour of functional programming [2],[12] but the fact is that functional programming suffers from a paramount lack of appreciation in the established programming society. For one reason this is caused by the fact that only recently functional programs have gained execution speeds comparable to their imperative rivals [13],[18],[8]. Another important reason is that functional programming defected on performing input output (I/O hereafter) with the outer world.

I/O and functional programming doesn't seem to be unifiable: functional languages don't have the notion of *side-effects* and lack a precise *control of the evaluation order*. These are precisely the important aspects for any I/O model. There have been many proposals how to deal with I/O (see section 8) but none of them are completely satisfactory. In particular it is very hard to deal with the side-effect problem.

In this paper we present Clean's *Event I/O*, a new solution for the I/O problem that has been implemented in the lazy functional graph rewriting language Concurrent Clean [5],[13],[17] developed at the University of Nijmegen. This solution not only deals with the problems mentioned above but it also allows the specification of complicated window based I/O on a very high-level of abstraction. Last but not least, the presented solution can be implemented efficiently and is general enough to be applied for other functional languages as well.

The paper is organised as follows. In section 2 we first give a very short description of Concurrent Clean. In section 3 we briefly explain Clean's Unique Types and explain how they can be used to define functions with side-effects. Furthermore we explain how explicit environment passing is used to control the evaluation order. Now we have the tools to change the world. In section 4 we show the world we have created with the I/O library. In section 5 we present as example the Game of Life to demonstrate the resulting high-level specification of Clean's Event I/O. The implementation of the I/O library is discussed in section 6. Section 7 discusses Clean Event I/O and it is compared with other solutions in section 8. Conclusions and future work can be found in section 9.

2 Concurrent Clean

Concurrent Clean [5],[13],[17] is a lazy functional programming language based on Term Graph Rewriting [4]. Here is an example of a Clean function defining the well-known fibonacci function.

```
:: Fib INT -> INT;
   Fib 1 -> 1;
   Fib 2 -> 1;
   Fib n -> + (Fib (- n 1)) (Fib (- n 2)),   IF  > n 2
         -> ABORT "Fib called with argument less than one";
```

Term Graph Rewriting systems are very suited for efficient implementations of functional languages [17]. Graph rewriting is actually used in many implementations. The main difference between Clean and other lazy functional languages is that in Clean graph rewriting is explicitly in the semantics of the language. In Concurrent Clean, the function application to be evaluated is represented by a possibly cyclic graph. Function definitions are actually Term Graph Rewriting rules. For instance, in the right-hand-side of the Fib definition above, actually a graph structure is defined. Each node in the graph contains a symbol (`+`, `Fib`, `-`, `1`) and arguments pointing to other nodes. In Clean, reasoning about programs is reasoning about graphs. It is straightforward to denote cyclic structures and shared computations. For instance, the argument node `n` is shared in the graph constructed on the right-hand side of the example reflecting the call-by-need evaluation of functional languages. Term graph rewriting obeys the functional semantics: given a rewrite rule which left-hand side matches the computation graph, a new graph is created for those nodes of the right-hand side which are new to the computation graph. After this, redirection to the new nodes takes place.

Concurrent Clean provides a type system based on the Milner/Mycroft scheme. There are a number of predefined types: **INT**, **REAL**, etc. and type constructors: lists **[]**, n-tuples **()** and curried functions **=>**. Furthermore there are algebraic types, synonym types and abstract types.

Clean has two types of modules: implementation modules and definition modules. The types and functions specified in an implementation module only have a meaning inside that module unless they are exported in the corresponding definition module. For more information we refer to [17] and [6].

3 Managing Side-effects in Concurrent Clean

In this section we explain how to enforce a specific order of evaluation and how to establish side-effects that are safe (i.e. retaining a pure functional language). First we will shortly introduce the general ideas which underlie the solutions we have found.

3.1 Basic Philosophy

Many of the proposed solutions (see section 8) to model I/O in functional languages have regarded I/O as something alien that in some way had to be incorporated. They all effectively do I/O 'outside' the program. Our starting point has been to do I/O 'inside' the program. We wanted to have a function that reads a character from a file, a function that writes a character to a file, a function that draws a line into a window, etc. So, we wanted to have functions that could change the state (and the contents) of some (abstract) object such as a file or a window.

Such "functions" are very common in imperative languages. There are two problems why this common solution cannot be applied in the functional paradigm. First of all, these functions are not proper functions: they actually perform a side-effect (assignment) to obtain the wanted effect: to change the contents of the abstract object. Assignments are not available in a functional language. The second problem is that functions can be evaluated in any order while functions that perform I/O have to be called in a very specific order.

3.2 Side-effects and Confluence

What kind of problems are caused by functions that perform side-effects? Take for example file I/O. The most obvious and efficient way to perform file I/O is by implementing functions that directly read and write to a file such as is common in imperative languages. However, a naive implementation of such functions in a functional language would conflict with the referential transparency. For instance, assume a function **FWriteC** that upon evaluation directly writes a character to a given file. Assume that such a function is of type **:: CHAR FILE -> FILE**. **FWriteC** takes a character and a file as an argument. However, the character cannot be written into the given file and returned as result because the original file can be shared and used in other function applications. Modification of the argument as a side-effect of the evaluation of a function will therefore also effect the outcome of other computations that share the same argument. The result of a program will now depend on the evaluation order, the Church-Rosser property is lost, the system is no longer confluent. This is illustrated in the following example:

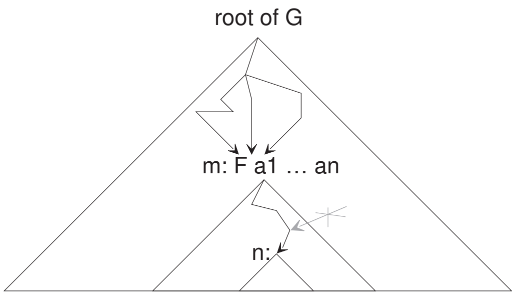
```
:: F FILE -> (FILE, FILE);
   F file -> (FWriteC 'a' file, FWriteC 'b' file);
```

Assume that the function `FWriteC` would actually append, as a side-effect, the given character to the file it receives as an argument. Now, since the file is shared in the function body of `F` the result will be depending on the evaluation order. It either will be `(file++'a', file++"ab")` or `(file++"ba", file++'b')`. And, indeed, such a side-effect is not conform the standard graph rewriting semantics that prescribes to construct a *new* file in the function body with the contents of the given file and the given character. So, each time a character is written a new file has to be constructed and the old ones have to remain intact. Now the result of `FWriteC` are two new files and the result then becomes `(file++'a', file++'b')` independent of the chosen evaluation order. Constructing new files is of course very inefficient and it is not the intention either. One really wants to have the possibility to modify (update) an *existing* file instantaneously. The problem becomes even more obvious when one wants to write to a window on a screen: one would like to be able to draw in an *existing* window. In the standard semantics one would be obligated to construct a new window with each drawing command.

3.3 Unique Types

Fortunately, side-effects *can* be allowed under certain conditions. *If* it can be guaranteed that the offered argument is not used by (shared with) other function applications it becomes garbage when it is not used in the function body. So, in that case one can construct a new object by making use of the old one. This means that one can destructively update such an argument to construct the function result. In Concurrent Clean, a type system is incorporated [6], [17] that guarantees that certain objects (unique objects) can be reused safely.

A node n of a graph G is **unique** with respect to a node m of G if n is only reachable from the root of G via m and there exists exactly one path from m to n. A property of a unique node is the fact that it has a reference count (in-grade) of one. A reference count of one is however not sufficient for uniqueness, the *whole* path from m to n must have reference count one.



Assume that a node is passed as argument of a certain function application in such a way that the node is unique with respect to that function application: if such a node is accessed via a variable in a pattern of the corresponding rewrite rule and that variable is not used on the right-hand side of that rule, it can be considered as garbage after matching and reused for building the function result. It would be nice if at compile time the uniqueness of arguments and results of functions could be determined. Unfortunately, this is undecidable. In Clean a decidable approximation has been incorporated using **unique types**. Unique types, defined on *graphs*, have many similarities with linear types, defined on *lambda terms* [7],[22]. An important difference is that Clean's unique types give information about the way *a specific function* has to be *applied* (e.g. this function has to be called with an argument that is used in a linear way) while other linear type systems give information about the way expressions are being used in the function body (e.g. this argument of the function is used linear in the function body).

The type of a graph in a rewrite rule can have the **unique type attribute**, i.e. the graph can be of type `UNQ T`. If a graph in a left-hand side or in the right-hand side of a rewrite rule is of type `UNQ T`, it is guaranteed that at run-time the root of the corresponding graph is unique with respect to the root of respectively the function application or function result (see [6], [17]).

The `UNQ` type attribute can be added by the programmer to *any* type to express the restricted use of an object of that type. To verify the correctness of the use of `UNQ` attributes the type system has been extended. This means that all applications on the right-hand side of a function are examined to check that when a parameter or a result of an `UNQ` type is demanded, a unique graph of the demanded type is offered.

An illustrative example is the following. The type of the function `FWriteC` that writes a character to a given file as a side-effect will be `:: CHAR UNQ FILE -> UNQ FILE`. This type specification guarantees that the function will always be called with an object of type `FILE` that is not used somewhere else. That is why the dangerous example given above becomes illegal:

```
:: F UNQ FILE -> (UNQ FILE, UNQ FILE);
   F file -> (FWriteC 'a' file, FWriteC 'b' file);
```

In the function body of `F`, `FWriteC` is in both applications not called with an object of type `UNQ FILE`, but an with an object of type `FILE` (because `file` is shared) and therefore rejected by the type system. The following example is approved:

```
:: F UNQ FILE -> UNQ FILE;
   F file -> FWriteC 'a' file;
```

It is OK to share unique elements. It simply means that the object is not unique anymore and therefore cannot be passed to functions that demand `UNQ` objects. So, the following definition is also fine (although not very useful):

```
:: F UNQ FILE -> (FILE, FILE);
   F file -> (file, file);
```

The `UNQ` type predicate is a powerful tool to enforce programs to use objects in a single threaded way. In Clean the `UNQ` type attribute can be assigned to any type, hence also be used in a user defined algebraic type.

3.4 Controlling the Evaluation Order

So, how can functions be evaluated in a specific order? A well-known method is **environment passing**. The state of the environment one wants to regard is coded into an (abstract) object (e.g. a file in the case of file I/O) on which operations (functions) are defined for creation and manipulation.

Each function that modifies the environment needs the current state of the environment as argument and yields the updated environment as result. In the case of file I/O this means that all functions that perform file I/O need a file as argument and return an updated file as result. So, the abstract object has to be explicitly passed from one function to another.

When a function performs an update of an argument as a side-effect, it must be guaranteed that all previous updates of that argument by other functions have already taken place. So, a function that updates an argument must be **hyper-strict** in this argument, i.e. it must be guaranteed that the argument is always in normal form before the function is applied. This means that functions that perform side-effects on an object that is passed around will evaluate this object in a fixed sequential order: innermost.

4 I/O

Existing environment passing schemes either pass the environment *explicitly* to all functions (explicit environment passing schemes) or *implicitly* to all functions (implicit environment passing schemes) as in the language FL [3]. In both schemes the environment is monolithic: it is one single structure and all access to parts of it must occur via the whole structure. The main disadvantages of using a monolithic environment are over specification of evaluation order and potential loss of parallelism.

In Clean, the Unique Type attribute can be assigned to *any* object. This gives us the possibility to create a "world" that is composed of disjunct sub-worlds. Each of these sub-worlds is an abstract data structure that can be uniquely passed around to those functions that need these sub-worlds to perform I/O. So, we have improved the explicit environment passing scheme by partitioning the monolithic environment in suitable sub-environments (`FILE` for file-I/O, `IOState` for screen-I/O). This avoids over specification of evaluation order and loss of parallelism. The Concurrent Clean I/O libraries offer the following environments to perform I/O: `WORLD`, `FILES`, `FILE` and `IOState`. These environments have the following hierarchic relationship:

```
                WORLD
          FILES
      FILE ... FILE        IOState
```

4.1 WORLD

The **WORLD** contains all relative information of the concrete environment to programs. In this version these are the concrete file system and the concrete event stream used for screen-I/O.

The **WORLD** is the only monolithic environment in Concurrent Clean. Programs specify their relationship to the world: pure computations ignore the world and its sub-environments, whereas interactive programs need to access and change the world. There are no rules creating **WORLD**s. The only way to get the world in the program is as argument of its initial rule, the **Start** rule. So, the start rule looks as follows:

```
ABSTYPE :: UNQ FILES;
RULE
:: Start WORLD -> any*type
   Start w -> any*computation*yielding*the*indicated*type;
```

Combined with the uniqueness of the **WORLD** it is guaranteed that there is *at most* one world in every program. Contrary to world realisations using linear types the **UNQ** typed world may become garbage during program evaluation. This does not mean that the world has ceased to exist, but it means that the program no longer performs operations on the **WORLD**. If the sub-environments have been retrieved earlier, then these can still be accessed in the program.

4.2 FILES and FILEs

FILES is the unique sub-environment of the **WORLD** containing all the files (the file system). It is a *unique abstract* type defined as follows:

```
ABSTYPE :: UNQ FILES;
RULE
:: OpenFILES WORLD -> (FILES, WORLD);
:: CloseFILES FILES WORLD -> WORLD;
:: FOpen   STRING INT FILES -> (BOOL, UNQ FILE, FILES);
:: SFOpen  STRING INT FILES -> (BOOL,     FILE, FILES);
:: FWriteC CHAR UNQ FILE -> UNQ FILE;
:: SFReadC FILE -> (BOOL, CHAR, FILE);
```

So, the file system is retrieved from the **WORLD** by the rule **OpenFILES** and put back again by the rule **CloseFILES**. Once the **FILES** has been retrieved from the **WORLD**, it can’t be retrieved again without closing it first.

A Concurrent Clean file has type **FILE**. To open a file (to read or write) one needs the file system. Only write **FILE**s are opened as **UNQ FILE**; read only **FILE**s don’t require the unique attribute. The following example illustrates the use of **WORLD**, **FILES** and **FILE**s:

```
RULE
:: Start WORLD -> UNQ FILE;
   Start w
   -> CopyF sf df,
      (fs, w'): OpenFILES w,
      (source_open, sf, fs' ): SFOpen "Source" FReadData  fs,
      (dest_open,   df, fs''): FOpen  "Dest"   FWriteData fs';

:: CopyF FILE UNQ FILE -> UNQ FILE;
```

```
CopyF sf df -> df,   IF NOT read_ok
             -> CopyF sf' (FWriteC char df),
                (read_ok, char, sf'): FReadC sf;
```

The program retrieves the **FILES** from the **WORLD**, after which the **WORLD** becomes garbage. From the **FILES** first the file to be copied is opened followed by the destination file. The resulting **FILES** also becomes garbage. The source file is only being read, so it need not be unique. The destination file is being written and must therefore be unique. After completion of copying, the source file becomes garbage, and the program yields the written file.

4.3 Event I/O

Event I/O is a *different* class of I/O than **FILE** I/O. In Event I/O the objects to be manipulated are graphical interface objects as windows, menus and dialogs. Graphical interface systems operate event driven: the user of a program communicates with that program via the interface objects: with the mouse one draws pictures, selects menu items, activates windows or presses radio buttons in a dialog. One uses the keyboard to fill in text fields in a dialog or to type text in an edit window, or to select menu items. These actions of the user generate *events* to the program. The operating system also uses events to communicate with the program to notify things have been changed. Finally, manipulations of the interface objects by the program may generate events as well. In sequential systems, these events are merged in one globally accessible event stream.

To program event driven applications one basically has to parse the events in the event stream such that the appropriate event handler can be called. However, this is very low level work which gives rise to rather ugly programs due to the complexity of event handling. In the Concurrent Clean Event I/O *all* low level event management is done in the library. The program reasons about interactions on a high abstraction level. In this level the concept of *Devices* is introduced. The aim of a Device is to capture the essence of its real life counterpart. A Device is an object with a consistent behaviour on a precisely defined set of input events. The semantics of the Device is partially fixed by the system, and can be partially specified by the program. Currently there are five devices: the WindowDevice, MenuDevice, DialogDevice, TimerDevice and NullDevice.

The WindowDevice is a real interactive device: its input domain is keyboard-presses and releases and mouse-clicks and releases coupled with the mouse position. The WindowDevice manages all open windows of an interaction. Interactions can have an arbitrary number of open windows. Of all these windows at most one is active: all keyboard events are directed to that window. An interaction can do screen output only by means of windows. A window gives a view on a Picture (an **UNQ** abstract object on which a set of drawing functions are defined). Pictures are finite objects: they have a range defined by the window's PictureDomain.

The MenuDevice conceptualises choosing commands from a set of available commands. A MenuDevice contains a number of pull down menus each holding a number of selectable menu items, sub-menus, menu item groups and menu radio items. Menu items are selected via the mouse or keyboard.

The DialogDevice models structured communication between program and user. The DialogDevice manages property and command dialogs, as well as notices. Property dialogs are always modeless, and are used to set properties of the interaction. Command dialogs can be modal or modeless. Property and command dialogs can contain editable, static and dynamic texts, radio buttons, check boxes, buttons (of arbitrary or standard shape) and program defined controls. Notices are very simple modal dialogs which are used to inform the user about unusual or dangerous situations. The dialog components are accessed by the user via the mouse or the keyboard.

The TimerDevice enables interactions to synchronise every specified time interval. The TimerDevice only responds to timer events. The NullDevice responds only to null events, special events that are generated in case there is no input.

Interactive programs can be regarded as specifications of state transition systems. One part of this state reflects the logical state of the interactive program, the *program state*. Every interaction defines its own program state. The only restriction on the program state is that it has an **UNQ** type. The other part of the state transitions concerns all devices the interaction wants to manipulate, grouped by the *IOState*. The IOState is the *unique abstract* sub-environment for programs that do event I/O. All actual event I/O the program performs, happens via the IOState: the devices themselves are never directly accessed.

```
ABSTYPE :: UNQ IOState UNQ s;

RULE
:: OpenIOState WORLD -> (IOState s, WORLD);
:: CloseIOState (IOState s) WORLD -> WORLD;
```

IOState is a *unique abstract* type. Analogous to **FILES** it is retrieved from the **WORLD** by the rule **OpenIOState** and put back again by the rule **CloseIOState**. Once an **IOState** has been retrieved from the **WORLD**, it can't be retrieved again without closing it first. The type variable **s** in **IOState s** is the program state **s** of the interaction. When retrieved from the **WORLD** the event stream is set into the IOState which still has to be filled with the devices of an interaction.

Each new event triggers a response of the program: given the event, the program state and the IOState, it is completely determined what the next program state and IOState will be. This implies that in order to fully specify an interaction, it is sufficient to define only the initial program state and the initial IOState. **IOSystem** is a large predefined algebraic type by which a program specifies *what* Devices will engage in an interaction; *how* these Devices will be *initialised* (the look) and which event handler (Clean function) has to be called.

Starting and terminating interactions is handled by two special functions: **StartIO** and **QuitIO**. **StartIO** takes the specification of the set-up of the I/O system as described above, the initial program state $\mathbf{s_0}$ and an **initial_io_state**. The initial I/O state can be obtained from the world by using the predefined function **OpenIOState**. The function StartIO will create the devices as specified and store the characteristics in an I/O state based on the initial I/O state. Then, it starts to 'poll' recursively waiting for input (events). The input event is dispatched to the proper device which computes the next program state and **IOState** by applying the proper event handler. In this way a sequence of pairs of program state and **IOState**, starting from the program state $\mathbf{s_0}$ and I/O state $\mathbf{IOState_0}$, is computed (below the implementation of **StartIO** is given using internal library functions).

```
StartIO io_system_specification program_state initial_io_state
-> DoIO (program_state, io_state),
   io_state: InitIO io_system_specification initial_io_state;

DoIO states:(program_state, ClosedIO_State) -> states;
DoIO (program_state, io_state)
-> DoIO new_states,
   new_states: event_handler event program_state io_state'',
   (event_handler, io_state''): GetHandler event io_state',
   (event,         io_state' ): GetEvent io_state;
```

The order of evaluation guarantees that the transition triggered by event e_{n+1} is only reduced after the transition triggered by e_n has yielded a complete $IOState_{n+1}$. The interaction obtained in this way can only be terminated by having any of the device functions apply **QuitIO** to their **IOState** argument. **QuitIO** produces a special IOState, **ClosedIO_State**, in which all devices are closed. **StartIO** matches on this special state and produces the final program state.

The program defined Device functions are of type **:: s (IOState s) -> (s, IOState s)** or **:: Event s (IOState s) -> (s, IOState s)** (with **s** the program state). This implies that during an interaction it is straightforward to create *nested* I/O of a completely different program state **t** by calling **StartIO** with an **IOSystem t**, an initial new program state **t** and the current **IOState s**. The currently running interaction is disabled and replaced by the new one until that interaction is terminated.

For the complete definition of the semantics of Clean File and Event I/O we refer to [1].

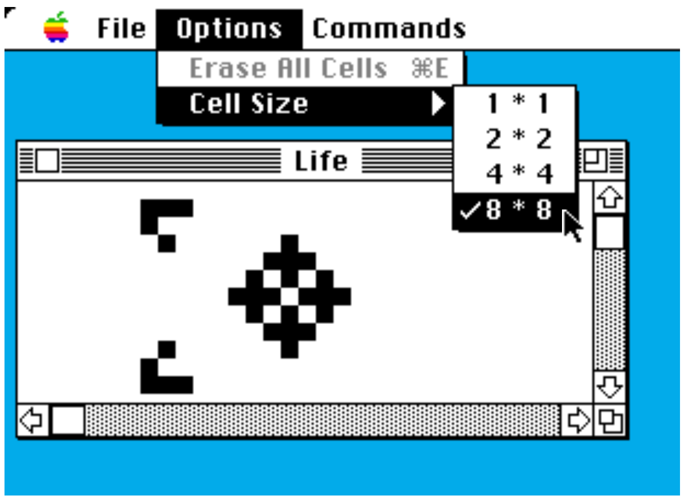
5 An Example: the Game of Life

In this section we present an example to illustrate a typical interactive Concurrent Clean program. The program describes the interface for a system playing the game of life. This is a 'game' consisting of an infinite two dimensional space (the universe). A cell is identified by a Cartesian position (**LifeCell**) in the universe. A cell is either alive or dead. An initial generation (**Generation**) of alive cells is sown. Each following generation is computed given the current one by two rules: only if an alive cell has two or three alive neighbour cells, it survives in the next generation, and only if a dead cell has three alive neighbour cells, it becomes alive in the next generation.

The module **Life** contains the rules that are not part of the interface. The rule **LifeGame** computes given a **Generation** (**[LifeCell]**) a triplet of the next **Generation**, new-born cells and died cells. **RemoveCell** removes and **AddCell** adds a **LifeCell** to a **Generation**.

The main module **LifeGame** describes the user interface. Apart from module **Life**, it imports the necessary Clean I/O modules and the **delta** modules for basic type computations. In general an interactive program has a type-block in which frequently occurring types are declared. In this program the program state is the tuple **UNQ State**, keeping the current **Generation** and the **Size** in which cells are displayed.

In the rule **StartIO** the **IOSystem** is defined. The interaction uses a **WindowDevice**, a **NullDevice** and a **MenuDevice**. The **MenuDevice** holds all commands the user of the program has at disposal. The menu item **Cell Size** is a sub-menu. Its elements **n * n** change the size of displayed cells into **n**. To warrant one size is valid at all times, the items are organised as **MenuRadioItems**. The initial selected size is **8 * 8**. Note that the items use the same function **SetSize** applied curried in its first argument. The **WindowDevice** manages only one window which ignores all input from the keyboard, but accepts all mouse events. Using the mouse a user can add or remove cells to the current generation. The **NullDevice** calculates the next generation when no action arises from the user. Observe the close relationship between the definitions of the window and menus and their visual appearance on screen.



The dynamics of the devices are in control by the **MenuDevice**: initially, the user is allowed to place cells in the universe. This means that the mouse of the window is enabled and the **NullDevice** disabled. As soon as the user selects **Play** the mouse is disabled and the **NullDevice** enabled. **Halt**ing the computations causes the **NullDevice** to be disabled and the mouse enabled.

```
MODULE LifeGame;

IMPORT delta, Life, deltaPicture;
IMPORT deltaEventIO, deltaMenu, deltaNull, deltaWindow;
```

This type-block defines the program state and a shorthand for the IOState.

```
TYPE
:: UNQ State -> (Generation, Size);
:: Size      -> INT;
:: UNQ IO    -> IOState State;

RULE
```

The execution of the program starts here. StartIO initialises the NullDevice, MenuDevice and WindowDevice. *Italic* rules in the device definitions are the program defined event handlers.

```
:: Start World -> (State, IO);
   Start world
   -> StartIO
        [WindowSystem windows,NullSystem null,MenuSystem menus]
        ([], 8) io_state,
      (io_state, world'): OpenIOState world,
      windows: [DocumentWindow LifeWindowId WindowPos "Life"
                     ScrollBarH&V ScrollBarH&V PictureRange
                     InitialSizeOfWindow MinimumSizeOfWindow
                     UpdateWindow [Mouse Able Track]],
      null   : Null Unable Step,
      menus  : [file, options, commands],
      file   : PullDownMenu FileMenuId "File" Able [
                 MenuItem QuitId "Quit" (Key 'Q') Able Quit],
      options: PullDownMenu OptionsMenuId "Options" Able [
                 MenuItem EraseId "Erase All Cells" (Key 'E')
                                                    Able Erase,
                 SubMenuItem CellSizeId "Cell Size" Able [
                   MenuRadioItems Size8Id sizes]],
      sizes: [MenuRadioItem Size1Id "1*1" NoKey Able (SetSize 1),
              MenuRadioItem Size2Id "2*2" NoKey Able (SetSize 2),
              MenuRadioItem Size4Id "4*4" NoKey Able (SetSize 4),
              MenuRadioItem Size8Id "8*8" NoKey Able (SetSize 8)],
      commands: PullDownMenu CommandsMenuId "Commands" Able [
                   MenuItem PlayId "Play" (Key 'P') Able   Play,
                   MenuItem HaltId "Halt" (Key 'H') Unable Halt];
```

Now all device functions are going to be defined. We start with the menu functions.

```
:: Quit State IO -> (State, IO);
   Quit state io -> (state, QuitIO io);

:: Play State IO -> (State, IO);
   Play state io
   -> (state, ChangeIOState [DisableMenuItems [PlayId, EraseId],
                             EnableMenuItems  [HaltId],
                             DisableActiveMouse,
                             EnableNullDevice] io);

:: Halt State IO -> (State, IO);
   Halt state io
   -> (state,
       ChangeIOState [DisableNullDevice,
                      EnableActiveMouse,
                      DisableMenuItems [HaltId],
                      EnableMenuItems  [PlayId, EraseId]] io);

:: Erase State IO -> (State, IO);
   Erase (gen, size) io
   -> (([],size),
       DrawInActiveWindow [EraseRectangle PictureRange] io);
```

SetSize draws the cells in the new size and changes the State accordingly.

```
:: SetSize Size State IO -> (State, IO);
   SetSize new (gen, cur_size) io
```

```
   -> ((gen, new),
       DrawInActiveWindow [EraseRectangle PictureRange |
                           Map (DrawCell new) gen] io);
```

The NullDevice computes the next generation and draws it.

```
:: Step State IO -> (State, IO);
   Step (gen,size) io
   -> ((next,size), ChangeIOState [DrawInActiveWindow erase,
                                   DrawInActiveWindow draw] io),
      erase: Map (EraseCell size) died,
      draw : Map (DrawCell  size) new,
      (next,new,died): LifeGame gen;
```

UpdateWindow draws all cells of the current generation regardless of their visibility.

```
:: UpdateWindow UpdateArea State -> (State, [DrawFunction]);
   UpdateWindow update_area state:(gen,size)
   -> (state,
       [EraseRectangle PictureRange | Map (DrawCell size) gen]);
```

Track is evaluated for all mouse-activities in the window.

```
:: Track MouseState State IO -> (State, IO);
   Track (pos, ButtonUp, modifiers) state io -> (state, io);
   Track ((x,y), down, modifiers) (gen,size) io,
      modifiers:(shift,option,command,control)
   -> ((remove,size), DrawInActiveWindow erase io),  IF command
   -> ((add   ,size), DrawInActiveWindow draw  io),
      remove: RemoveCell cell gen,  erase: [EraseCell size cell],
      add   : AddCell    cell gen,  draw : [DrawCell  size cell],
      cell  : (/ (- x (% x size)) size, / (- y (% y size)) size);
```

Drawing or erasing a cell in a given cell size.

```
:: DrawCell Size Cell -> DrawFunction;
   DrawCell size (x,y)
   -> FillRectangle ((px: * x size, py: * y size),
                     (+ px size,    + py size));

:: EraseCell Size Cell -> DrawFunction;
   EraseCell size (x,y)
   -> EraseRectangle ((px: * x size, py: * y size),
                      (+ px size,    + py size));
```

The program constants to enhance comprehension and maintenance.

```
MACRO
FileMenuId -> 1;    OptionsMenuId -> 2;    CommandsMenuId -> 3;
   QuitId -> 11;       EraseId -> 21;         PlayId -> 31;
                         CellSizeId -> 22;    HaltId -> 32;
                         Size1Id -> 221;
                         Size2Id -> 222;
                         Size4Id -> 223;
                         Size8Id -> 224;
LifeWindowId -> 1;       MinimumSizeOfWindow -> (50,   50);
WindowPos    -> (0,0);   InitialSizeOfWindow -> (1000, 1000);
PictureRange -> ((0,0), (1000,1000));
ScrollBarH&V -> ScrollBar (Thumb 400) (Scroll 8);
```

6 Implementation on Macintosh and X Windows

A few remarks are worth mentioning about the implementation of Concurrent Clean event I/O on Macintosh and X Windows systems.

The I/O system has been designed in such a way that the program only specifies *what* interaction has to be done, never *how*. Interactions are specified in terms of the algebraic data structure IOSystem. All event handling and device handling has been carefully hidden from the program. This approach has made it possible to implement the Concurrent Clean event I/O system on Macintosh and X Windows systems, two quite different systems. Both implementations have been done from the specification down to the respective interfaces. This confirms the suitability of the chosen abstraction level of the Clean devices.

Because interactions are specified in terms of Clean objects, it was possible to write major parts of the library code in Clean, making extensive use of higher order functions (approximately 50% Clean code for the X Windows implementation and 90% for the Macintosh implementation). All device definitions, `IOState` and `Picture` are defined in Concurrent Clean as (abstract) algebraic types. The implementation on the Macintosh uses a thin interface layer which contains Clean rules for most procedures of the Macintosh toolbox. The implementation on X Windows [16] uses an interface layer to the Open Look™ Interface Toolkit (olit) and the general X Windows libraries.

The advantages of this approach are that using algebraic types for all objects of the I/O system allows the use of higher-order functions in the object definitions. Device definitions are ordinary functional objects and can be manipulated the same way as other functional objects. It is easy to experiment with various kinds of devices and system maintenance has become easier.

7 Discussion

We have experienced that Clean's Unique Type system is a very useful tool. We did not have any problems writing the library in Clean itself (5000-7000 lines of code) with the restrictions imposed by the use of unique objects.

With this library many interactive applications have been written in Clean among which some very large ones (a complete Mac-style copy-paste editor and a DBase-like relational database). Due to the high-level of abstraction offered by the I/O-library, it is possible to write compact reliable device independent interactive functional programs in a relatively short time period. The libraries maintain the different look-and-feel of the specific machines.

Of course, not every thinkable bell and whistle has been predefined in the library. But, most of the commonly used I/O gadgets are available and there is a possibility to specify user-defined controls. Furthermore, the library is well structured such that new facilities and devices can be added relatively easy.

Also the large applications written in Clean (together with the library over 10.000 lines of Clean code) still run efficient enough to be used in practice (even on small machines). They behave as good as (or sometimes even better than) their imperative counterparts. It reveals something about the code generated by the Clean compiler as well as about the quality of the libraries of present day object oriented languages.

There are two negative things to be mentioned. Clean programs consume more memory than the imperative programs (typically four times as much). Furthermore, there are some classical and non-classical language features missing in the present version of Concurrent Clean (Clean was designed as intermediate language, not as a full flavoured functional language). We want to solve both problems in the future.

8 Related Work

Other solutions to deal with I/O in functional languages can be divided in a number of categories, each having typical advantages and disadvantages. The (implicit) environment passing approach as taken in the language FL [3] has as main disadvantage that the order of evaluation is fixed. This is problematic for program transformations, introducing sharing of computations and exploitation of parallelism.

The token stream approaches [21],[9],[10],[14] suffer from inefficiency, because programs are obliged to get input by outputting tokens (doing requests). The input can be handled synchronously or asynchronously. When it is handled asynchronously one has to parse the input to determine the corresponding response. When it is handled synchronously an additional synchronisation overhead is introduced. The advantage of synchronised token streams is the control over the domain of input tokens because the program has specified from which devices the input will originate.

With the interaction type of solutions [20],[11] one has to take care that output and input occurs in the right order. The solution to this problem as given by [20] is a predefined set of functions which behave correctly in this respect.

A nice solution is the use of a predefined monad [15] which guarantees the single threaded use of predefined functions with side-effects. A disadvantage is that all functions with side-effects have to be applied on one and the same (hidden) monad. So, for instance it is not possible to define a hierarchy of sub-environments.

9 Conclusions and Future Work

Concurrent Clean's Event I/O provides programmers with a very high-level declarative specification method for writing complex interactive applications in a pure high-order functional language. All low-level I/O handling is done automatically. The library offers most of the commonly used I/O facilities and can easily be extended with new devices and facilities. The device oriented approach yields concise and elegant programs which are easy to understand and maintain.

Currently there is a version of the I/O library for the Sun under X-Windows/Open Look and a version for the Macintosh. These libraries provide the same interface such that a Clean program can run on either of these systems without any modification. Still, the resulting applications will obey the different look-and-feel which is typical for these machines. The library has been used to write several applications. We obtain very good runtime performances even compared with imperative programs.

In the future we want to extend the I/O model for distributed environments. The Unique Type system as well as Clean itself will be refined further to increase flexibility and user-friendliness. Except for type checking, the current Clean system does not yet exploit the full potential of the **UNQ** type predicate, which is a very interesting research for improving the efficiency of Clean programs in both time and space [19].

References

1. Achten PM. Operational Semantics of Clean Event I/O. Technical report - in preparation University of Nijmegen.
2. Backus J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.In: Communications of the ACM, Vol.21 Nr.8, 1978.
3. Backus J, Williams J, Wimmers E. An introduction to the programming language FL. In: Turner A (ed) Research topics in Functional Programming, Addison-Wesley Publishing Company, 1990.
4. Barendregt HP, Eekelen van MCJD, Glauwert JRW et al. 'Term Graph Reduction'. In: Proceedings of Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, LNCS 259, Vol.II. Springer-Verlag, Berlin, 1990, pp. 141-158.
5. Brus T, Eekelen van MCJD, Plasmeijer MJ and Barendregt HP. Clean - A Language for Functional Graph Rewriting. In: Proc. of Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, Springer Verlag, LNCS 274, 1987, pp. 364-384.
6. Eekelen van MCJD, Huitema HS, Nöcker EGJMH, Smetsers JEW and Plasmeijer MJ. Concurrent Clean Language Manual - version 0.8. Technical report No.92-18 Department of Informatics, Faculty of Mathematics and Informatics, University of Nijmegen 1992.
7. Girard J-Y. Linear Logic. In: Theoretical Computer Science 50. 1987, pp. 1-102.
8. Groningen van JHG, Nöcker EGJMH and Smetsers JEW. Efficient Heap Management in the Concrete ABC Machine. In: Proc. of Third International Workshop on Implementation of Functional Languages on Parallel Architectures. University of Southampton, UK 1991. Technical Report Series CSTR91-07.
9. Hudak P. et al (ed) Report on the Programming Language Haskell, - A Non-strict, Purely Functional Language -, Version 1.1 (as made public available in August 1991).
10. Darlington P. Purely Functional Operating Systems. In: Darlington, Henderson, Turner (ed) Functional programming and its applications.
11. Dwelly A. Functions and Dynamic User Interfaces. In: Proc. of ACM 1989. pp. 371-381.
12. Hughes J. Why Functional Programming Matters. In: Turner DA (ed) Research topics in Functional Programming. Addison-Wesley Publishing Company, 1990.
13. Nöcker EGJMH, Smetsers JEW, Eekelen van MCJD and Plasmeijer MJ. Concurrent Clean. In: Proc. of Parallel Architectures and Languages

Europe, Eindhoven, The Netherlands. Springer Verlag, LNCS 505, 1990, pp. 202-219.

14. Perry N. Functional I/O - a solution. Department of Computing, Imperial College, London, Draft version, July 1988.
15. Peyton Jones SL, Wadler Ph. Imperative Functional Programming. Extended Abstract, to appear in POPL 1993, University of Glasgow.
16. Pillich L. Portable Clean Event I/O. Department of informatics, Faculty of Mathematics and Informatics, University of Nijmegen. Master Thesis 230, July 1992.
17. Plasmeijer MJ, Eekelen van MCJD. Functional Programming and Parallel Graph Rewriting. Lecture notes. University of Nijmegen 1991/1992. To appear: Addison Wesley 1993.
18. Smetsers JEW, Nöcker EGJMH, Groningen van JHG and Plasmeijer MJ. Generating Efficient Code for Lazy Functional Languages. In: Proc. of Conference on Functional Programming Languages and Computer Architecture Cambridge, MA, USA, Springer Verlag, LNCS 523, 1991, pp. 592-617.
19. Smetsers JEW, Achten PM, Eekelen van MCJD and Plasmeijer MJ. An Unique Type Predicate and its Application for Efficient Code Generation for Functional Languages. Technical report - in preparation. University of Nijmegen.
20. Thompson S. Interactive Functional Programs. A Method and a Formal Semantics. In: Turner DA (ed) Research topics in Functional Programming, Addison-Wesley Publishing Company, University of Kent, 1990.
21. Turner DA. An Approach to Functional Operating Systems. In: Turner DA (ed) Research topics in Functional Programming, Addison-Wesley Publishing Company, University of Kent, 1990.
22. Wadler Ph. Linear types can change the world! In: Broy M, Jones CB (ed) Programming Concepts and Methods, North-Holland, 1990.