2
Single Interactive Functional Programs

Functional programming languages have banned assignment because of its undesirable properties. The reward of this rigorous decision is that functional programming languages are side-effect free. There is another side to the coin: because assignment plays a crucial role in input/output (I/O), functional languages have a hard time dealing with I/O. Functional programming languages have therefore often been stigmatised as inferior to imperative programming languages because they cannot deal with I/O very well. In this chapter we show that I/O can be incorporated in a functional programming language without loss of any of the generally accepted advantages of functional programming languages. This discussion is supported by an extensive account of the I/O system offered by the lazy, purely functional programming language Clean. Two aspects that are paramount in its I/O system make the approach novel with respect to other approaches. These aspects are the technique of *explicit multiple environment passing*, and the *event I/O framework* to program graphical user I/O in a highly structured and high-level way. Clean file I/O is as powerful and flexible as it is in common imperative languages (one can read, write, and seek directly in a file). Clean event I/O provides programmers with a high-level framework to specify complex graphical user I/O. It has been used to write applications such as a window-based text editor, an object based drawing program, a relational database, and a spreadsheet program. These graphical interactive programs are completely machine independent, but still obey the look-and-feel of the concrete window environment being used. The specifications are completely functional and make extensive use of uniqueness typing, higher-order functions, and algebraic data types. Efficient implementations are present on the Macintosh, Sun (X Windows under Open Look), and PC (OS/2).

2.1 Introduction

Functional programming languages live in a world from which assignment (or destructive updating) has been banned because of its undesirable properties. Living without assignment has proven to be very successful, and many accounts have been written of the advantages of living in a world free of side-effects (Backus, 1978; Hughes, 1990). However, in order to write useful applications, it must be possible for functional programs to interact with the outside world. Doing I/O means manipulation of I/O resources, such as files, keyboards, mice, and screens. In the real world these resources are globally accessible, and manipulations of them are in essence assignments. This implies that functional languages cannot use I/O resources in the same direct, unrestricted way as for example imperative languages can. For this reason functional languages are often stigmatised as inferior to imperative programming languages because they cannot deal with I/O very well.

Research on the incorporation of purely functional I/O into functional programming languages has evolved into basically two styles of solutions: *stream* based solutions and *environment* based solutions. *Stream* based methods have been proposed in a (token) stream style (Henderson, 1982; Turner, 1990; Hudak *et al.,* 1992; Carlsson and Hallgren, 1993) and continuation style (Thompson, 1990; Dwelly, 1989; Perry, 1988). Essentially, stream based methods transform an *input stream* into an *output stream*. The output stream is not exclusively used for producing output only, it is also used for requesting input. Some entity outside the program (usually the operating system) handles the output requests and provides the proper input. *Environment* based methods are environment passing methods (Williams and Wimmers, 1988; Backus *et al.,* 1990) and methods using monads (Peyton Jones and Wadler, 1993). In these solutions functions essentially operate directly on a special object, the *environment*, that represents the *state of the world*. In the literature environment based methods are also known as *side-effecting* I/O systems (Gordon, 1993).

The Clean I/O system that is presented in this chapter is an environment based approach and contributes to the research in functional I/O in two major aspects. The first aspect is the use of an *explicit multiple environment passing* style throughout the system giving explicit and direct access to I/O resources. This has been made possible by the *uniqueness type system* of Clean (Smetsers *et al.,* 1993; Barendsen and Smetsers, 1993*a-b*; Plasmeijer and van Eekelen, 1993) which enables safe and restricted updates in a pure and functional framework. The second aspect provides programmers with the Clean event I/O framework (Achten *et al.,* 1993; Achten and Plasmeijer, 1993) to program graphical user I/O in a highly structured and declarative way. The specifications of interactive programs are functional and programs can be reasoned about without any assumption about operating systems. The I/O system demonstrates that functional languages are well suited for I/O, by making extensive use of uniqueness typing, and well-known functional programming features such as higher-order functions, polymorphism, and algebraic types.

The chapter starts with brief introductions to Clean and uniqueness types (Sections 2.2 and 2.3). The explicit multiple environment passing style is defined in Section 2.4. Clean file I/O is discussed in Section 2.5, and Section 2.6 presents the Clean event I/O system. Section 2.7 discusses how interactive programs can be constructed in the Clean event I/O system, and Section 2.8 briefly views the implementation of the Clean event I/O system. Section 2.9 compares some related work with our approach. Finally the conclusions are presented in Section 2.10, and current and future research on functional I/O is presented in Section 2.11.

2.2 Clean

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). To give an idea of what Clean programs look like, Figure 2.1 presents an example of the well-known fibonacci function. The examples in this chapter are presented in the new Clean 1.0 syntax (Plasmeijer and van Eekelen, 1994, *in preparation*). Where appropriate, the text includes remarks on peculiarities of this notation.

```
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)

Start :: Int
Start = fib 100
```

Figure 2.1 A Clean program for fibonacci. Function definitions are optionally preceded by their type definition. Type symbols start with a capital, (type) variables always start in lowercase. Function names can start either with a capital or in lowercase. An n-ary function named `f` with arguments of type $\tau_1 \ldots \tau_n$, and result type τ has a type definition `f` :: $\tau_1\ \tau_2 \ldots \tau_n \rightarrow \tau$. The special function named `Start` gives the initial expression of the program.

Term graph rewriting systems are well suited for efficient implementations of functional languages (Groningen *et al.,* 1991; Smetsers *et al.,* 1991; Plasmeijer and van Eekelen, 1993). Graph rewriting is actually used in many implementations of functional languages. 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 Clean, the function application to be evaluated is represented by a possibly cyclic computation graph. Function definitions are actually term graph rewriting rules. Each rule alternative is a graph with a left-hand side root (L.H.S.) and a right-hand side root (R.H.S.). Figure 2.2 depicts the graph structure of the third `fib` alternative. Each node in the graph contains a *symbol* (`fib`, `+`, `-`, `1`, `2`) and *arguments* pointing to other nodes.

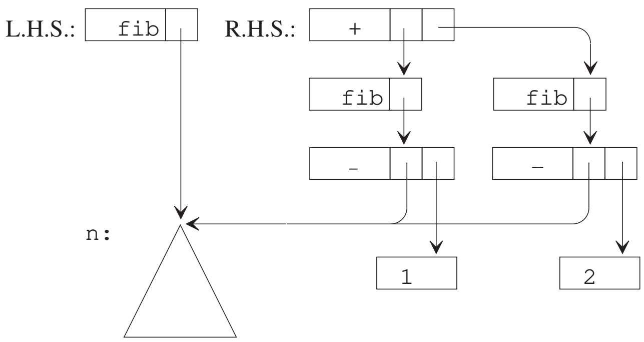


Figure 2.2 The third alternative of the fibonacci rule depicted as a graph.

In Clean, reasoning about programs is reasoning about computation graphs. It is straightforward to denote cyclic structures and shared computations. For instance, the semantics of Clean prescribe that the argument node `n` is shared in the computation graph constructed on the right-hand side of the example explicitly reflecting the call-by-need evaluation which is commonly used in the actual implementation of functional languages.

Term graph rewriting obeys the functional semantics. Figure 2.3 illustrates one formal rewrite step of the computation graph `fib 100` of the fibonacci example (the implementation is done in a much more efficient way!). The initial graph (*a*) consists of only one redex, namely the graph `fib 100`, which matches the third alternative of the `fib` rewrite rule. Rewriting this redex occurs in the following way: a new graph is created for those nodes of the right-hand side of the rule that are new to the computation graph (in the example these are two nodes labelled `fib`, and nodes labelled `+`, `−`, `1`, and `2`). This process is called graph extension (*b*). After extending the computation graph, the original computation graph root is overwritten with the root of the extended graph that matches the right-hand side of the rule (*c*). In term graph rewriting terminology this process is called 'redirection' of the left-hand side root to the right-hand side root. Finally, the nodes that have become unreachable from the new root of the computation graph are garbage collected (*d*).

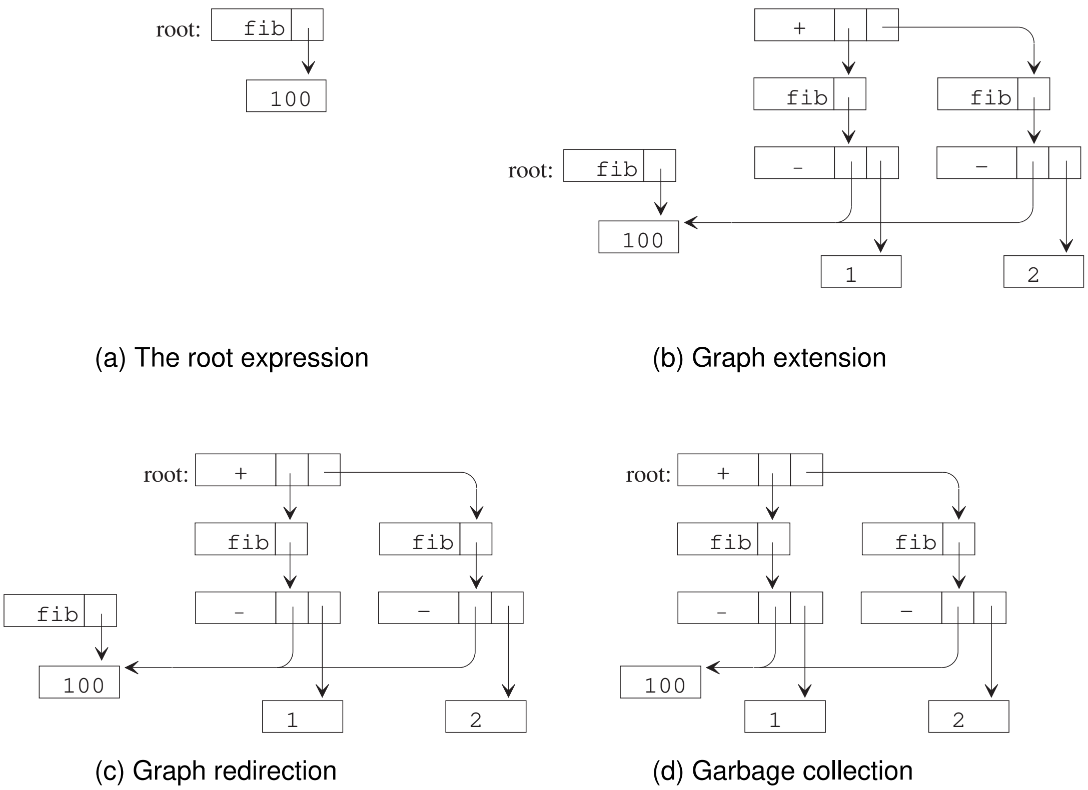


Figure 2.3 One rewrite step of the initial expression of the fibonacci example.

In general, a computation graph consists of several redexes. The rewriting process needs a *reduction strategy* to determine what redex should be rewritten. The default reduction strategy of Clean is the lazy functional strategy. A (sub)graph that contains no redex is said to be in *normal form*. A (sub)graph in which the root node is not a function symbol is said to be in *root normal form*. In the remainder of this chapter when we discuss Clean we will use the term *functions* for *rewrite rules* and vice versa for convenience.

2.3 Uniqueness types

Because Clean is based on a typed term graph rewriting system it is possible in this system to use type information to state properties of graphs. One such interesting property states that a specific subgraph of a computation graph is not shared by any other node of that graph. A subgraph that fulfils this property is said to appear *uniquely* in the computation graph. More formally the uniqueness property is stated as follows (Plasmeijer and van Eekelen, 1993):

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*.

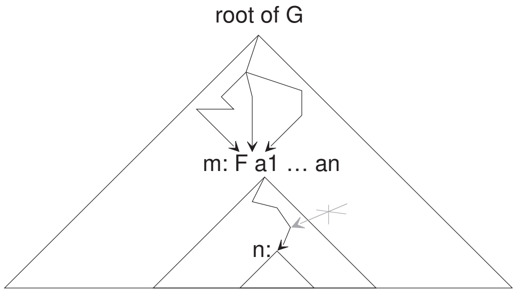


Figure 2.4 The uniqueness property depicted.

Why is this an interesting property? To answer this question, it is necessary to recall the rewriting semantics of Clean. In this system, rewriting a matching rule alternative in a computation graph (the redex) creates a completely new graph matching the right-hand side of the rule alternative. The redex root is redirected to the newly created graph. If we know that an offered argument of this rule is unique with respect to the application node and it is not used in the function body then it will become *garbage*. In that case a new object can be constructed by making use of the old one. This means that one can destructively update such an argument to construct the function result. If the offered argument of the rule is not known to be unique with respect to the application node then it is illegal to reuse the argument because it might be shared.

It would be nice if the uniqueness of arguments and results of functions could be determined at compile time. Unfortunately, this is undecidable. In Clean a decidable approximation has been incorporated using *uniqueness types* (Smetsers *et al.,* 1993; Barendsen and Smetsers, 1993*a-b*). Uniqueness types differ from linear types (Girard, 1987; Wadler, 1990-*a*) defined on lambda calculus. An essential difference is that, in the analysis of uniqueness types, graphs play a crucial role. Uniqueness types restrict the use of graphs and function applications in a program, whereas linear types restrict the use of variables inside function definitions. The relationship between uniqueness types and linear types is a topic of further investigation. Closer related work to the uniqueness type system is by Guzmán and Hudak (1990) who present an extended lambda calculus with state operations which safety is warranted by the type system.

The uniqueness type system is quite complex, and a formal treatment of it is beyond the scope of this chapter. The complete formal framework of uniqueness types can be found in Barendsen and Smetsers (1993-*a*), the main results of this work have been published in Barendsen and Smetsers (1993-*b*). The incorporation of this formal type system in Clean is described in Plasmeijer and van Eekelen (1994, *in preparation*). For this chapter it is sufficient to know that the uniqueness attribute '*' can be assigned to any type (*synonym* types, *algebraic* types, and *abstract data* types) by prefixing the attribute to the type.

The uniqueness type system uses a kind of reference count analysis called *sharing analysis*. The sharing analysis allows an arbitrary number of references to a unique object as long as it can statically deduce that the reference count will be one when the object is accessed by the function that wants it to be unique. The sharing analysis marks each reference in a right-hand side as *not-shared* (if it could be shown that the object points to a not-shared object) or *shared* (otherwise). There are several cases to mark a reference not-shared. In case there is only one reference in the right-hand side of a rule to a certain object (the reference count of the object is one) the mark clearly should be not-shared. If it can be shown that the evaluation order is such that other references will be vanished on time, they are not counted and the reference to be marked will still be marked as not-shared. An example of such a situation is the reference to an object in both a guard and its guarded expression. The guard will be evaluated before the guarded expression will be evaluated, so the reference is lost when the guarded expression is evaluated. As a result, unique objects are allowed to be *observed* in guards.

Objects marked shared by the sharing analysis cannot be typed unique. So, the sharing analysis is input for the type system to check uniqueness type consistency. For each reference (argument in a node) it is determined how many other references there will exist whenever the object is accessed (and evaluated to root normal form) via this reference. The type checker verifies the correctness of the use of uniqueness attributes in rules by examining all applications on the right-hand side of a function to check that when a parameter or a result of a uniqueness type is *demanded*, a unique graph of the demanded type is *offered*. In this context demanded type means that either the corresponding formal parameter with the applied function has a uniqueness attribute or the result type with the defined function has the uniqueness attribute.

The uniqueness types system is a powerful tool which provides a wide range of interesting applications in the implementation and use of functional languages. It provides the basis of efficient and functional I/O, it can be used for the implementation of destructively updateable arrays and user-defined unique data structures, it can be used in the analysis of memory usage of functional programs (as has been done by Chirimar *et al.,* 1992 for a language based on linear types), and it can serve as a general safe interfacing facility for functional languages with the imperative world.

2.4 Explicit multiple environment passing

Specifications of interactive programs require a method in which sequences of I/O operations can be defined. In order to be able to reason properly about interactive programs it is vital that these sequences be evaluated in a predictable order (e.g.: in the teletype kind of interactive systems prompts must appear before one waits on user input) and that the evaluation of an I/O operation has an immediate effect (when the prompt is demanded to appear it must show on the screen). In this section we introduce an improved type of explicit environment passing scheme that will provide these vital properties for a lazy functional language. This scheme is used throughout the I/O system.

2.4.1 Explicit environment passing

Explicit environment passing schemes are well suited as methods for specifications of interactive programs. In an explicit environment passing scheme there is one special data object in normal form, the *environment*, which is some sort of encoding of (changes in) the state of the world. A program doing I/O is a function that given an initial environment produces a new environment in which all subsequent changes are contained. Programs can change the state of the world and retrieve information from the world by functions that have access to this environment. The evaluation of such a function consists of two actions: the state of the world is changed immediately, and a new instance of the environment is yielded in which the change to the state of the world is reflected. We will call the change to the state of the world the *effect* of the function. Because we intend these functions to have an immediate effect, they are *hyper-strict* in their environment arguments. As a result the environment argument will always be in normal form before the function is evaluated. Functions that inspect (read) the world yield what has been read. So, every access rule accounts for its effect on the state of the world in the environment object. Each new operation is applied to the environment result of the previous operation. Sequences of operations are easily expressed as sequences of function applications on the environment.

Figure 2.5 gives a small example of what a typical explicit environment passing program looks like. Suppose we have an environment of type `World`. The function `echo` is a simple recursive function on `World`. It retrieves a character from the environment by some predefined function `getchar` and prints it on screen using some predefined function `putchar`. The recursion of `echo` terminates if a newline character, denoted with '`\n`', has been retrieved by `getchar`.

```
echo :: World -> World
echo world
|    c = '\n'      = world2
|    otherwise     = echo world2
where
     (c,world1)    = getchar world
     world2        = putchar c world1
```

Figure 2.5 An example of the explicit environment passing style. Guarded expressions are preceded by a conditional expression |.

The explicit environment passing style can be seen by the way `world` (which is a value of type `World`) is used to pass around the state of the world after each operation. The effect of the program is rather obvious: if a user types the character sequence $c_1 \ldots c_n$'\n' (all characters c_i are not the newline character), then the screen will show the character sequence $c_1 c_2 \ldots c_n$. Moreover, the program expresses successfully that each character c_i is being put on screen immediately after it has been read and before character c_{i+1} is being read.

This clear use of explicit environment passing schemes makes them very attractive as a basis for a functional I/O system with direct access to I/O resources. The idea of using explicit unrestricted environment passing schemes is indeed not new. Gordon (1993) mentiones the unpublished PhD. thesis of Redelmeier (1984) in which this idea is presented. However, there is a catch to unrestricted explicit environment passing. The environment represents the world and as there is only one world around one gets into serious problems as soon as the environment is duplicated or shared. Sharing the environment allows the introduction of an arbitrary number of environment changing sequences. The manipulations on the world that are performed in one sequence are not recorded in the environments of the other sequences. Because the world has been updated according to some interleaving of these manipulations none of the resulting environment objects reflect the state of that updated world anymore.

The program in Figure 2.6 illustrates this catch. The function `catch` does two things on the world: put a newline on screen (by `world1`), and echo the keys typed by a user of the program using `echo` from the previous example (by `world2`). Suppose the user of this program types the same character sequence $c_1 \ldots c_n$'\n' as previously. The output of the program can be any character sequence $c'_1 c'_2 \ldots c'_{n+1}$ with for some i ($1 \leq i \leq n+1$), c'_i='\n', $c'_j = c_j$ (for $j<i$), and $c'_j = c_{j-1}$ (for $j>i$) because the order of evaluation of `world1` and `world2` is undetermined. So the state of the real world contains the character sequence $c'_1c'_2...c'_{n+1}$. However, environment `world1` records a real world with state '\n', and `world2` records a real world with state $c_1...c_n$'\n'. Neither environment correctly reflects the state of the world.

```
catch :: World -> (World, World)
catch world
=   (world1,world2)
where
    world1 = putchar '\n' world
    world2 = echo world
```

Figure 2.6 A program illustrating the danger of unrestricted environment passing.

Despite the catch, Clean I/O is based on the explicit environment passing style. The environment that is passed around in the Clean I/O system is of type `*World`. The initial environment is given as an optional argument of the `Start` rule. It should be noted that the environment object cannot be introduced by a function because it would introduce the possibility to introduce an arbitrary number of environment objects. The only proper way to deal with the world is to regard it as a parameter of a program. In order to avoid the catch, and to reflect the 'unique' nature of the actual world represented by the environment, all environment operations require their environment argument to have the uniqueness attribute (so the example functions have the types `getchar` :: `*World` → (`Char`, `*World`) and `putchar` :: `Char` `*World` → `*World`). Due to the uniqueness typing the Clean I/O system restricts access of the program to the environment, and prevents sharing and introduction of multiple environments. The type system of Clean rejects `catch` due to the fact that `putchar` demands `world` to have type `*World`, but instead `world` has offered type `World` (because `catch` contains two references to `world`). Obviously, the offered type cannot be coerced to obtain the uniqueness attribute. The function `echo` needs a small addition in its type definition to turn it into a correctly typed Clean definition (see Figure 2.7).

```
echo :: *World -> *World
echo world
|   c == '\n'   = world2
|   otherwise   = echo world2
where
    (c,world1) = getchar world
    world2     = putchar c world1
```

Figure 2.7 The function `echo` now as a correctly typed Clean program.

2.4.2 Multiple environments

Environment passing schemes based on one single environment enforce programmers to create a spine of I/O function applications in a program. This is a very severe restriction on functional program expressiveness, as programs are obliged to over determine order of evaluation. To our knowledge, this is basically true for all safe environment based approaches in functional I/O (see also the discussion in Section 2.9), and also for all stream based approaches (as they consider one single stream that carries the I/O operations).

Reconsider, for example, the `echo` function in Figure 2.7. In this program the spine of I/O operations is formed by the sequence `getchar`, `putchar`, `getchar`, `putchar`,... of read/write operations. However, for this program it is sufficient to express that at least as many characters are read as there are characters printed. This relationship cannot be defined in an environment passing scheme without fixing an evaluation order.

The combination of explicit environment passing and uniqueness types is a powerful one as it allows a very liberal and safe use of *multiple* environments. Introducing multiple environments allows a program to define multiple sequences of I/O operations without predetermining an evaluation order between these sequences. Other advantages of using multiple environments are that such sequences of applications can be evaluated in parallel, and environments can be used to support modular programming of interactive programs.

The Clean I/O system defines a *hierarchy of environments* (see Figure 2.8). Therefore in our terminology environment should not be understood as an encoding of the state of the world as a whole but rather as a specialised data structure that encodes the state of a *specific* part of the world. These environments must be independent: operations on one environment should not have an effect on another environment. In the Clean environment hierarchy the top environment is the environment of type `*World`. Two sub environments can be retrieved from the world environment by *decomposition*. One represents the state of the *file system* and the other represents the *event stream* communication to and from graphical user interface elements. Their corresponding types are `*Files` and `*Events` respectively.

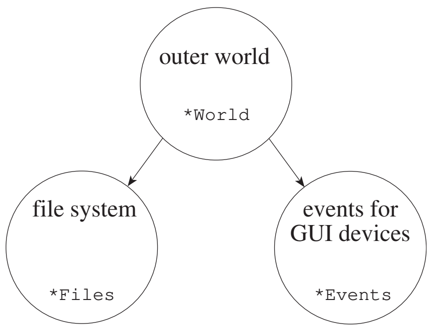


Figure 2.8 The Clean environment hierarchy.

The decomposition rule `openworld` of the world environment into the file system and event stream environments has type `*World` → (`*Files`, `*Events`). It should be noted that as a result the world is no longer available for subsequent use. The environments can be used in the program and finally compose a new world again, by a *composition* rule `closeworld` of reverse type (`*Files`, `*Events`) → `*World`. The start rule of a Clean program that does I/O is always of type `*World` → `*World`. In this way environments that have contributed to the effect of the program are always restored to the world environment. This is called *hygienic use* of environments.

2.5 File I/O

Clean file I/O is a good example of an I/O system using the explicit multiple environment passing scheme. The top environment of the file I/O system is the object of type `*Files` introduced in the previous section that encodes the real world file system. This environment is again a container of yet smaller environments: the individual files themselves (see Figure 2.9).

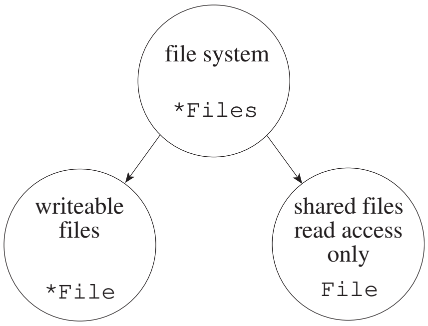


Figure 2.9 The Clean environments for file handling.

A Clean file has type `File` or `*File`. To open or close a file one needs a unique file system. Writeable files are opened as `*File`; read only files do not require the uniqueness attribute. Once a writeable file has been opened it cannot be opened again until the file is closed. Read only files can be opened an arbitrary number of times, but cannot be opened as writeable files anymore. Because read only files do not change the state of the file system they do not need to be closed, but can be made garbage safely when they are not needed anymore.

All this is controlled and administrated by the unique file system which is needed for the opening and closing of all files. It should be noted that the unique file system models the actual file system. All the file administration is in reality handled by the operating system. This implies that there is no need whatsoever to administrate anything in Clean itself. This means that all file I/O is handled as efficiently as possible because there is no administration overhead in the functional implementation component. The use of these files is as powerful, flexible, and efficient as it is in common imperative programming languages. For instance, in both types of files (`File` as well as `*File`) it is possible to perform random access (seeks). The Clean file primitives allow all basic types to be written directly to files and read from files. One can write to and read from writeable files in any order.

The programming style when using files is basically the explicit environment passing style. Figure 2.10 shows an example of a file copying program that illustrates the use of `World`, `Files` and `File`. The first action of the program is to decompose the unique `world` into the file system `files` and event stream `events`. The file system is used to open the source and destination files (first `source` is opened for reading, using the predefined function `sfopen`, and then `dest` is opened writeable by `fopen`). As `source` is going to be read only it is opened as a shareable file. The file `dest` is being written into and must therefore have the uniqueness attribute. The function `copyFile` copies the contents of `source` character by character to `dest`. After completion of copying, the written file is closed in the file system, and the final world is composed from the file system and the event stream.

```
Start :: *World -> *World
Start world
=   world1
where
    (files,events)     = openworld world
    (_,source,files1) = sfopen "Source" FReadData  files
    (_,dest,  files2) = fopen  "Dest"   FWriteData files1
    dest1              = copyFile source dest
```

```
    files3            = fclose dest1 files2
    world1            = closeworld (files3, events)

copyFile :: File *File -> *File
copyFile source dest
|   readOK    = copyFile source1 (fwritec c dest)
|   otherwise = dest
where
    (readOK,c,source1) = sfreadc source
```

Figure 2.10 A program copying a file named `Source` to a file named `Dest`.

2.6 Graphical user I/O

The techniques involved in programming graphical user interfaces make an interesting area of research because the corresponding I/O (*graphical user I/O*) is radically different from file I/O and is much more complicated. In a graphical user interface system the graphical user I/O is done entirely with graphical user interface elements such as *windows*, *menus*, and *dialogues*. These interface elements are characterised by a highly interactive behaviour. Applications that use graphical user interface systems have a very dynamic use of interface elements. Graphical user interface systems are *event driven*. An event is a data object recording a true event in the outside world or the operating system. Events come from different sources: the user of a program communicates with that program via interface objects in the course of which events are generated (e.g.: key presses, mouse movements). The operating system uses events to communicate to the program that things have been changed (e.g.: windows become partially visible, programs are scheduled). Finally, manipulations of the interface objects by the program may generate events as well (e.g.: opening and closing of windows or dialogues). The operating system provides these events for programs by the so-called event stream. The event stream is a sequence of events. In this sequence event A precedes event B if, and only if, A has occurred before B.

The Clean event I/O system is the framework a program uses to do graphical user I/O. The Clean event I/O system is an *abstract graphical user interface*. In the Clean event I/O system graphical user I/O is defined entirely with *abstract devices*. Abstract devices are abstractions of categories of concrete interface elements. The Clean event I/O system provides four abstract devices: the *window* device, *menu* device, *dialogue* device, and *timer* device. Abstract devices are specified on a high level of abstraction making extensive use of *algebraic types*. Abstract device specifications define which graphical user interface elements are used by the program, and how these elements interact with the user or other elements. Clean event I/O programs are *abstract event driven*. An abstract event is always defined in the context of an abstract device element. Clean event I/O programs do not retrieve abstract events, but rather define *abstract event handlers*. An abstract event handler is a function that is included in the abstract device specification. Only when the corresponding abstract event occurs the abstract event handler is evaluated. The abstract event handler is applied to the current *state* of the program and yields a new state. The state of the program consists of the data the program needs at run-time, and the run-time state of its interface elements. A Clean event I/O program has access to its interface elements at run-time via a special unique environment of type `*IOState`. A Clean event I/O program only needs to specify the abstract devices to create an interactive program. The Clean event I/O system takes care that the abstract devices are correctly mapped to the concrete devices, and that the concrete events are correctly mapped to abstract events.

In Section 2.6.1 we show how abstract devices and its interface elements are defined using algebraic types. Section 2.6.2 focusses on abstract event handlers, and explains how these functions affect the run-time state of the program and the interface elements. Section 2.6.3 defines how the Clean event I/O system uses abstract device definitions, and the appropriate environments to create a running program. Section 2.6.4 briefly describes the abstract devices other than the menu device. Finally, Section 2.6.5 gives a small example of an interactive program.

2.6.1 Defining abstract devices

Abstract devices provide Clean programmers with a high level view of graphical user interface elements. These abstract interface elements are specified by functional expressions that are instances of a set of predefined algebraic types (see Figure 2.11 and 2.12). For each abstract device there is defined an algebraic type that fully specifies how the individual interface elements of that abstract device should be defined. Because the algebraic types contain functions that have to operate on the same type of program state, the type definitions are parameterized with the type variable *s* which reflects the type of the program state.

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

Figure 2.11 The algebraic type definition of devices. The type [α] is a list of α. The symbols printed in **boldface** are alternative constructors of the algebraic type (variants of the type).

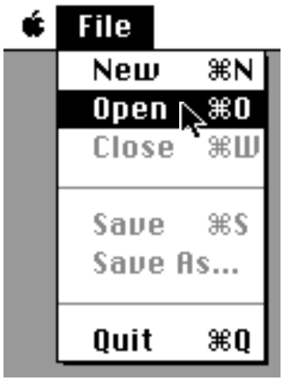
```
::  MenuDef     s
=   Menu        Title [MenuElement s] [MenuAttribute s]
::  MenuElement s
=   SubMenuItem Title [MenuElement s] [MenuAttribute s]
|   MenuItem    Title                 [MenuAttribute s]
|   MenuSeparator
::  MenuAttribute    s                      // Default:
=   MenuId           Id                     // no Id
|   MenuSelectState  SelectState            // menu(item) Able
|   MenuShortKey     KeyCode                // no KeyCode
|   MenuAltKey       Index                  // no AltKey
|   MenuMarkState    MarkState              // NoMark
|   MenuFunction     (IOFunction     s)     // λx.x
|   MenuModsFunction (ModsIOFunction s)     // λxy.y

::  IOFunction     s :==              s -> (IOState s) -> (s,IOState s)
::  ModsIOFunction s :== Modifiers -> s -> (IOState s) -> (s,IOState s)
```

Figure 2.12 The algebraic type `MenuDef` to define individual menus. The types `IOFunction` and `ModsIOFunction` are synonym types.

As an illustration of an abstract device definition, Figure 2.13 gives an example of a menu definition. The picture below the definition shows the concrete device in the case of the menu definition being mapped to a Macintosh system.

```
Menu "File" [
  MenuItem "New"       [MenuFunction New,   MenuShortKey 'n'],
  MenuItem "Open…"     [MenuFunction Open,  MenuShortKey 'o'],
  MenuItem "Close"     [MenuFunction Close, MenuShortKey 'w',
                        MenuSelectState     Unable          ],
  MenuSeparator,
  MenuItem "Save"      [MenuFunction Save,  MenuShortKey 's',
                        MenuSelectState     Unable          ],
  MenuItem "Save As…" [MenuFunction SaveAs,
                        MenuSelectState     Unable          ],
  MenuSeparator,
  MenuItem "Quit"      [MenuFunction Quit,  MenuShortKey 'q']
  ] []
```

Figure 2.13 An example of a menu definition in Clean.



Algebraic types prove to be very useful as a medium for abstract device definitions in a functional language for several reasons.

(**1**) In a functional language it is trivial to add the abstract event handlers to algebraic types because functions are 'first-class citizens' and can be used in a curried way. For instance, the menu definition in Figure 2.13 specifies that the program code that should be evaluated when the menu item titled "Open…" is selected is the function named `Open`. This is the simple case. But it is also possible to define a higher-order function applied to an arbitrary number of arguments as abstract event handler, which is messy to realise in the classical imperative languages.

(**2**) Algebraic types provide a *specification language* of which the syntactical correctness is verified by the type checker. This eliminates obvious programming errors (like typing errors, or mixing up order of arguments of different type) that occur rather frequently in text-based specifications.

(**3**) The use of algebraic types for all abstract device specifications provides both the programmer as well as the definition of the semantics with a *formal notation*. A formal notation is invaluable to disambiguate discussions on the meaning of individual interface elements.

(**4**) Algebraic type definitions can be made very intelligible and suggestive. We have carefully chosen suggestive names for the data constructors of the algebraic type definition of abstract devices which match their actual appearances as much as possible (WYSIWYS:

What You Say Is What You See). This is clearly illustrated by the example in Figure 2.13.

(**5**) Finally, readable definitions of interface elements serve as good *documentation* of programs.

2.6.2 Abstract event handlers

Clean event I/O programs are driven by abstract events that are defined in the context of abstract devices. Consider for example the menu definition in Figure 2.13. One abstract event defined in the context of this definition is *the menu item named 'Open…' has been selected*. It is easy to correlate the abstract event with an abstract event handler, because the abstract event is defined in the context of the algebraic definition. Therefore it is sufficient to add the abstract event handler in the context of the abstract device definition that defines the abstract event. So the response of the program to the abstract event is given by the function `Open`.

A Clean event I/O program consists of a number of abstract devices, which in turn define the set of possible abstract events, and their corresponding abstract event handlers. It is not determined in what order the abstract events will occur. Each abstract event handler can be evaluated in any state of the program. In order to handle the abstract event appropriately the abstract event handler needs to know the state of the program. As a result the state of the program will have been changed. So, an abstract event handler is a state transition function. The state of the program is a data object which has a fixed type (not a fixed value) because any of the available abstract event handlers must be applicable.

The state of the program consists of a component controlled by the programmer, and a component controlled by the Clean event I/O system. The program controlled component, called the *program state*, contains the data the program needs during evaluation. The program state can have an arbitrary, but uniquely attributed type. The component controlled by the Clean event I/O system is an abstract data type object which contains the run-time states of the interface elements of the program. This component is a uniquely attributed environment that is specially created for doing graphical user I/O. The type of the environment is `*(IOState s)` (it is a parameterized type because this environment also contains the abstract event handlers, which types are based on the program state of type `s`).

Abstract event handlers change the state of the program. So the types of abstract event handlers are of the form `.s *(IOState .s)` → `(.s, *IOState .s)`. (Note that `*(IOState .s)` and `(*IOState .s)` denote the same type. In case brackets are not required `*IOState .s` is also an equivalent notation.) With the abstract event handlers a programmer defines how the state of the program should be affected in case the abstract event handler is triggered by an abstract event. Changes on the program state component can be easily defined by the programmer, because this component is defined by the programmer. The `IOState` environment is an abstract data object, so changes on this environment can only be done via a library of predefined functions, the *abstract device access functions*. All device access functions take the explicit environment passing style. Their types are of the form $\tau_1 \ldots \tau_n$ `*(IOState .s)` → (τ, `*IOState .s`) or of the form $\tau_1 \ldots \tau_n$ `*(IOState .s)` → `*IOState .s`. For example, typical operations on the menu interface elements at run-time are enabling and disabling the entire menu system (also of separate menus or menu elements), adding and removing menu elements to and from menus, marking menu elements, and changing titles or abstract event handlers of menu elements (see Figure 2.14). Like all environment operations, every abstract device access function changes the appropriate interface element resources, and records the effect in the new `IOState` environment.

```
OpenMenu              :: Int [MenuDef .s] *(IOState .s) -> *IOState .s
CloseMenu             :: Id               *(IOState .s) -> *IOState .s
EnableMenuSystem      ::                  *(IOState .s) -> *IOState .s
DisableMenuSystem     ::                  *(IOState .s) -> *IOState .s
EnableMenus           :: [Id]             *(IOState .s) -> *IOState .s
DisableMenus          :: [Id]             *(IOState .s) -> *IOState .s
OpenMenuItems         :: Id Int [MenuElement .s]
                                          *(IOState .s) -> *IOState .s
CloseMenuItems        :: [Id]             *(IOState .s) -> *IOState .s
CloseMenuIndexItems   :: Id [Int]         *(IOState .s) -> *IOState .s
EnableMenuItems       :: [Id]             *(IOState .s) -> *IOState .s
DisableMenuItems      :: [Id]             *(IOState .s) -> *IOState .s
MarkMenuItems         :: [Id]             *(IOState .s) -> *IOState .s
UnmarkMenuItems       :: [Id]             *(IOState .s) -> *IOState .s
SetMenuItemTitles     :: [(Id,Title)]     *(IOState .s) -> *IOState .s
SetMenuItemFunctions  :: [(Id,MenuAttribute .s)]
                                          *(IOState .s) -> *IOState .s
```

Figure 2.14 The types of the menu device access functions of the Clean event I/O system.

2.6.3 Interactions

In Section 2.4.2 we have introduced the event stream environment. The previous two sections presented how graphical user interface elements are defined by abstract devices, discussed abstract event handlers, and introduced the state of a program. In this section we show how these elements of the Clean event I/O system are integrated by the *interaction concept* to obtain a running interactive program. This relationship is roughly illustrated by the following equations:

abstract device definitions + event stream = `IOState`
`IOState`+ program state = interaction

An interaction is a *dynamic state transition system* where the *transitions* are defined by the abstract event handlers of the abstract device definitions and where evaluation is triggered by the occurrence of abstract events. The abstract device definitions of an interaction are gathered in a single data object of synonym type :: `IOSystem s` :== [`DeviceSystem s`] (see Figure 2.11 for the type definition of `DeviceSystem`). The library function `OpenIO` evaluates an interaction given initial abstract device definitions of type `IOSystem`, the initial program state, the initial actions of the interaction of synonym type `InitIO` (:: `InitIO s` :== [(`s`, `IOState s`) → (`s`, `IOState s`)]), and the event stream environment:

```
OpenIO :: (IOSystem .s) .s (InitIO .s) *Events -> (.s, *Events)
```

`OpenIO` performs three actions: (**1**) creation of the proper environments for the interaction, (**2**) evaluation of the initial actions of the interaction, and (**3**) the evaluation of the interaction until termination.

(**1**) `OpenIO` is provided with the definitions of the abstract devices that will participate in the interaction. With these abstract device definitions the concrete graphical user interface elements are created. As a result the abstract devices appear to the user in their initial run-time state. The environment of type `*IOState` is filled with the run-time states of the concrete devices and their abstract event handlers.

(**2**) The second action of `OpenIO` is the evaluation of the initial actions of the interaction. Suppose the initial actions are the list of transition functions $[f_1 \ldots f_n]$, and the initial interaction state is the pair (s_0, io_0). The result of the initial actions is the new pair $(s_1, io_1) = f_n \circ f_{n-1} \circ \ldots \circ f_1\ (s_0, io_0)$ where $\circ$ denotes function composition. The initial actions are very convenient for an interaction to do initialisation actions such as setting up files, verification procedures, and so on (it may even decide to quit the interaction by applying the function `QuitIO` to its `IOState` environment).

(**3**) Finally, `OpenIO` evaluates the interaction until termination. This is done by an *event loop*, which is a simple, recursive function. In each step it retrieves a concrete event from the event stream environment. If the concrete event should be interpreted as an abstract event, the corresponding abstract event handler is applied to the current interaction state (s_i, io_i) to obtain the new current interaction state (s_{i+1}, io_{i+1}). The effect of this transition (which is administrated in io_{i+1}) is paired with the concrete event that triggered the transition. The event loop *terminates* as soon as the `IOState` component of the interaction state contains no more concrete devices. The result of `OpenIO` is the final program state and the changed event stream environment which contains the pairs of concrete events that have been parsed by the interaction and their effect, and the concrete events that have not been parsed by the interaction.

An interaction that decides that it should terminate, can do so by removing all concrete devices from its current interaction state. This can be done only with the library function `QuitIO` with `QuitIO :: *(IOState .s) → *IOState .s`. `QuitIO` releases the run-time resources of each device in the interaction and removes them from the `IOState` component.

2.6.4 The other abstract devices

In this section we briefly discuss the abstract devices other than the menu device. Their complete algebraic type definitions and abstract device access functions can be found in the Appendices A.3 upto A.7.

The timer device

The timer device enables interactions to synchronise on an arbitrary number of time intervals of arbitrary length. Timing is handled by assuming that all events are provided with a time stamp. This mechanism cannot provide real-time timing because the time needed to evaluate an abstract event handler may exceed a given time interval. The abstract event handler of every active timer is therefore provided with the discrete number of complete time intervals that have passed. The accuracy of the timer device is adequate for most animation tasks, or checks that need to be done at a regular basis.

The window device

Windows are the basic medium in which graphical user interface systems communicate with users. An interaction can have an arbitrary number of windows open. Windows are *stack ordered* which means that they can overlap. Of these windows at most one window is *active*. The active window is the window that receives all keyboard and mouse events. The interaction as well as the user can decide which window to activate. The active window is not required to respond to user events: the addition of a keyboard and mouse event handler is optional. Windows can change the *mode* of an interaction: when opened, a *modal* window forces the user of the interaction to deal with the window entirely before any other actions can take place. *Modeless* windows are less demanding: the user may disregard them and use them when convenient.

Windows can either be *general* windows or *dialogue* windows. Each general window has a *local drawing environment* of type `*Picture`. Pictures have finite dimensions, given by the `PictureDomain`. The application presents information to its users in a window by drawing into it. Both kinds of windows can have an arbitrary number of controls. These controls are common elements such as (radio) buttons, (edit) text fields, pop up elements, and check boxes but also user defined custom controls or compound controls. As a consequence, the definition of a set of controls is more complicated than the definition of a menu because the *layout* between controls needs to be defined. In order to ease the effort of defining dialogues, controls have a *layout attribute* of algebraic type `ItemPos` to influence their position (see Appendix A.1.1). With this layout attribute positions can be expressed relative to the size of the window (`LeftTop`, `RightTop`, `LeftBottom`, `RightBottom`, `Left`, `Center`, `Right`), and relative to earlier placed items (`LeftOf`, `RightTo`, `Above`, `Below`).

2.6.5 Example

In this section we present an example program that uses several abstract devices in order to illustrate how to program with the Clean event I/O libraries. The program is a simple address database. The user of the program can add and remove addresses to and from the database, view the current list of addresses, and quit the application. The end of this section contains the main fragment of the program code which contains the data structures and the abstract device definitions. The implementation of some functions have been omitted for reasons of clarity. Figure 2.15 gives a snapshot of the application running on a Macintosh system.

The main data structure in the program is the program state. The program state is the record type `AddressBook`. It contains the current list of address records (field `addresses`), an index to the currently selected record (field `selection`), a Boolean that records whether the list has been changed (field `saveChanges`), and the unique file system (field `files`). Because it contains the unique file system, the program state is also uniquely attributed. The initial program state contains no address records, no selection, no change of the list, and the file system retrieved from the `World` environment. The initial actions of the interaction are to read the address data base file, and to open the windows named "Addresses" and "Edit Address" that allow the user to alter the current contents. An address record, of type `Address`, contains the person's name, city, street, and telephone number.

In Clean 1.0, a record type is an algebraic type with exactly one alternative constructor. The alternative constructor does not need to be specified if the field names uniquely identify the record type. Record types and record expressions always appear between '{' '}' in a program. The arguments of a record expression can be selected by an extended form of *pattern-matching*, and by the *field names* of the arguments. On a pattern-match position, the expression 'r=:{x=y}' binds the variable r to the record value, and the variable y to the value of field 'x'. The shorthand notation 'r=:{x}' binds the variable x to the value of field 'x'. The expression 'r.x' denotes the value of field 'x' of the record value r. Entirely new records are denoted by summarizing all fields and binding them to values. Given a record value r with field 'x', the expression '{r & x=y}' denotes the new record with the same value r, and the field 'x' with value y. Finally, the standard library function `seq` is defined by seq [] x = x, and seq $[f_1 \ldots f_n]$ x = $f_n \circ \ldots \circ f_1$ x.

The initial abstract device of the application consists of one single menu. This menu, named "Book" defines the four available commands `Open`, `Add`, `Delete`, and `Quit`. `Open` opens the "Addresses" window in which the current list of addresses is shown. `Add` opens the "Edit Address" window by which the user can add addresses to the database. `Delete` removes the currently selected address from the database. This command is initially disabled because at start the user has not selected an address. `Quit` terminates the application. If the address list has been changed it also asks the user if changes to the database should be saved to the database file.

In the "Edit Address" window the user can fill in the text fields of an address. Pressing the button control "Add" triggers the abstract event handler `AddAddress`. This function retrieves the EditControl contents from the window, and creates an address record which is inserted in the current list of addresses. It increases the picture domain of the "Addresses" window, and updates the contents of the window.

Address records are selected manually by the user by pressing the mouse button when the mouse pointer is in the text area of that address in the "Addresses" window. These mouse actions are handled by the mouse event handler `Select`. It determines which address is selected (the value index), enables selection of the `Delete` command, unhighlights the previously selected address (index by the field selection of the program state), highlights the selected address, and fills the text fields of the "Edit Address" window with the field values of the selected address.

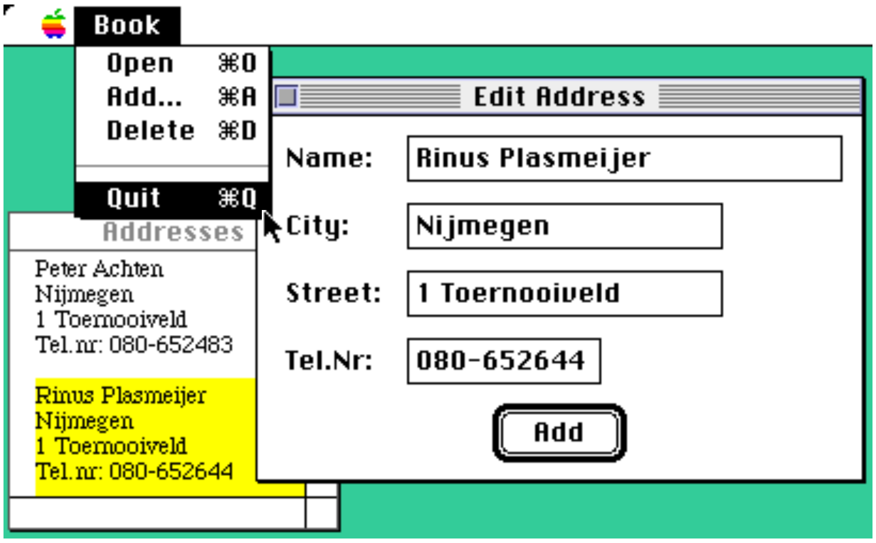


Figure 2.15 A snapshot of the address database application while running on a Macintosh system.

```
module addressBook

import StdEnv, StdFile
import StdEventIO, StdWindow, StdControl, StdMenu
import StdFont, StdPicture, StdIOState


::  *AddressBook  =   { addresses   :: [Address],
                        selection   :: Int,
                        saveChanges:: Bool,
                        files       :: *Files   }
::  Address       =   { person      :: String,
```

```
                        city       :: String,
                        street     :: String,
                        telNr      :: String  }
::  *IO            :== IOState AddressBook

AddressWindowId    :== 1
AddDialogId        :== 2
NameId             :== 1
CityId             :== 2
StrtId             :== 3
TelNrId            :== 4
DeleteId           :== 1

readaddresses      :: *Files -> ([Address],*Files)
writeaddresses     :: ([Address],*Files) -> *Files
```

read/write the list of addresses from/to the database file

```
remove             :: Int     [Address] -> [Address]
insert             :: Address [Address] -> [Address]
```

remove/add the indicated entry from/to the list of addresses

```
sliderControl      :: Direction Size -> ControlDef *AddressBook
```

definition of the control that defines scrolling in the window

```
calcPictureDomain :: [Address] -> PictureDomain
```

determine the coordinate system of the window given this list of displayed addresses

```
hiliteaddress      :: Int Picture -> Picture
```

toggle the hilighting of the indicated element

```
Start :: *World -> *World
Start world
=   finalWorld
where
    (fs, es)   = openworld world
    initBook   = {addresses=[],selection=0,saveChanges=False,files=fs}
    initIO     = [initAddressBook]
    (book,es1) = OpenIO [MenuSystem [menu]] initBook initIO es
    finalWorld = closeworld (book.files, es1)
    menu = Menu "Book" [
             [MenuItem "Open"   [MenuShortKey 'o',MenuFunction Open],
              MenuItem "Add…"   [MenuShortKey 'a',MenuFunction Add ],
              MenuItem "Delete" [MenuShortKey 'd',
                                 MenuFunction     Delete,
                                 MenuId           DeleteId,
                                 MenuSelectState  Unable],
              MenuSeparator,
              MenuItem "Quit"   [MenuShortKey 'q',MenuFunction Quit]]
             []

initAddressBook :: *AddressBook *IO -> (*AddressBook, *IO)
initAddressBook book io
=    Add (Open {book & addresses=addrs, files=fs} io)
where
    (addrs,fs) = readaddresses book.files

Open :: *AddressBook *IO -> (*AddressBook, *IO)
```

```
Open book=:{addresses} io
=   (book, OpenWindow window io)
where
    window = Window "Addresses" domain
                 [sliderControl Horizontal size,
                  sliderControl Vertical   size]
                 [WindowId     AddressWindowId,
                  WindowMouse  Able Select,
                  WindowSize   size,
                  WindowResize (\_ _ book io -> (book,io)),
                  WindowClose  (seqPIO [CloseWindow AddressWindowId])]
    domain                      = calcPictureDomain addresses
    ((left,top),(right,bottom)) = domain
    size                        = (right-left, bottom-top)

Select :: MouseState *AddressBook *IO -> (*AddressBook, *IO)
Select ((_,y),ButtonDown,_) book=:{addresses,selection} io
|   isEmpty addresses  = (book, io)
|   otherwise          = ({book & selection=index}, io1)
where
    io1 = seq [EnableMenuItems [DeleteId],
               DrawInWindow AddressWindowId [hiliteaddress selection],
               DrawInWindow AddressWindowId [hiliteaddress index],
               SetWindow    AddDialogId
              [SetEditTextControl NameId  address.person,
               SetEditTextControl CityId  address.city,
               SetEditTextControl StrtId  address.street,
               SetEditTextControl TelNrId address.telNr]] io
    address      = addresses!dec selection
    index        = y/itemHeight+1
    itemHeight   = 5*AddressBookStyle.height
Select _ book io = (book, io)

Add :: *AddressBook *IO -> (*AddressBook, *IO)
Add book io
=   (book, OpenWindow add io)
where
    add = DialogWindow "Edit Address"
            [CompoundControl [TextControl "Name:"   [],
                              TextControl "City:"   [left],
                              TextControl "Street:" [left],
                              TextControl "Tel.Nr:" [left]] look [],
             CompoundControl
               [EditControl "" 200 1 [ControlId NameId],
                EditControl "" 200 1 [ControlId CityId, left],
                EditControl "" 200 1 [ControlId StrtId, left],
                EditControl "" 150 1 [ControlId TelNrId,left]] look [],
             ButtonControl  "Add"    [ControlId addId, center,
                                      ControlFunction AddAddress]]
            [WindowId    AddDialogId,
             WindowClose (seqPIO [CloseWindow AddDialogId]),
             WindowOk    addId]
    left     = ControlPos (Left,  (0,0))
    center   = ControlPos (Center,(0,0))
    addId    = 1+max [NameId,CityId,StrtId,TelNrId]
    look _ _ = []

    AddAddress :: *AddressBook *IO -> (*AddressBook, *IO)
```

```
    AddAddress book=:{addresses,selection} io
    =   SetPictureDomain AddressWindowId domain book1 io1
    where
        (_,window,io1)= GetWindow AddDialogId io
        address    = {person = snd (GetEditTextControl NameId  window),
                      city   = snd (GetEditTextControl CityId  window),
                      street = snd (GetEditTextControl StrtId  window),
                      telNr  = snd (GetEditTextControl TelNrId window)}
        addresses1 = insert address addresses
        domain     = calcPictureDomain addresses1
        book1      = {book & addresses=addresses1,saveChanges=True}

Delete :: *AddressBook *IO -> (*AddressBook, *IO)
Delete book=:{selection,addresses} io
=   SetPictureDomain AddressWindowId domain book1 io1
where
    addresses1 = remove (selection-1) addresses
    domain     = calcPictureDomain addresses1
    book1      = {book & addresses  = addresses1,
                         selection  = 0,
                         saveChanges= True}
    io1        = DisableMenuItems [DeleteId] io

Quit :: *AddressBook *IO -> (*AddressBook, *IO)
Quit book=:{saveChanges} io
|   not saveChanges = quit book io
|   otherwise       = OpenModalWindow notice (book, io)
where
    notice = DialogWindow ""
               [TextControl   "Save changes to address file?"
                                  [ControlPos      (Center, nul)],
                ButtonControl "No"  [ControlFunction quit,
                                     ControlPos      (Left,(0,20))],
                ButtonControl "Yes" [ControlFunction yes,
                                     ControlId       yesId]]
               [WindowOk yesId]
    nul       = (0,0)
    yesId     = 1
    yes book=:{addresses,files} io
              = quit {s & files=writeaddresses (addresses,files)} io
    quit s io = (s, QuitIO io)
```

2.7 Structuring interactive programs

In many cases interactive programs can be decomposed into a number of distinct interactive units. For instance, many applications offer users a facility to edit text. Instead of programming these facilities over and over again for each new application, one would like to write a text editing module once, and include it in some way in various applications. In this section we will first show how interactions can be used as interactive modules, and then proceed with individual abstract device elements.

In the Clean event I/O system, interactions can be combined sequentially, or nested. Interactions are sequentially combined by function application: the event stream result of an application of `OpenIO` is the argument of the second application of `OpenIO`. Figure 2.16 gives an example of sequential interaction composition. The function `seqIO` when applied to two interaction definitions A and B, first evaluates interaction A and then interaction B. For notational convenience, we introduce a synonym type :: `IODef s` :== (`IOSystem s, s, InitIO s`) that collects the interaction definition components.

```
seqIO :: (IODef .s) (IODef .t) *Events -> *Events
seqIO (systemA,progStateA,ioA) (systemB,progStateB,ioB) events
=   eventsB
where
    (_,eventsA) = OpenIO systemA progStateA ioA events
    (_,eventsB) = OpenIO systemB progStateB ioB eventsA
```

Figure 2.16 Sequential composition of interactions. The _ symbol is a wild card which is a convenient denotation for anonymous node identifiers.

Interactions can be nested with the library function *`NestIO`*. Any interaction can start the evaluation of a new interaction during its own evaluation.

```
NestIO:: (IOSystem .s) .s (InitIO .s) *(IOState .t) -> (.s,*IOState .t)
NestIO system progState io iostate
=   (progStateN, iostate4)
where
    iostate1             = HideIO iostate
    (events,   iostate2) = getEvents iostate1
    (progStateN,events1) = OpenIO system progState io events
    iostate3             = setEvents events1 iostate2
    iostate4             = ShowIO iostate3
```

The type of `NestIO` is similar to the type of `OpenIO` except for being applied to the `IOState` environment of the running interaction rather than the event stream environment. The type `s` of the program state of a nested interaction is in general different from the type `t` of the program state of the *parent* interaction that starts the nested interaction. The nested interaction is completely evaluated and only after its termination the parent interaction continues evaluation. `NestIO` takes care that before the nested interaction takes over from the parent interaction, the parent is hidden (applying function `HideIO`). This means that all the visible graphical user interface elements of the parent interaction disappear from screen, and cannot be accessed by the user. After termination of the nested interaction, the parent interaction is shown again (applying the function `ShowIO`). As a result, all interface elements of the parent interaction that had been hidden from the user reappear. Interactions can be nested arbitrarily deep and arbitrarily many. Figure 2.17 gives an example of a function `editLine` that provides a nested text editing facility.

```
editLine :: String *(IOState .s) -> (String, *IOState .s)
editLine line io
=   (getEditLine s1, io1)
where
    (s1,io1) = NestIO SystemEdit (InitEditState line) InitIOEdit io
```

Figure 2.17 A text editing interaction that can be used in arbitrary interactions.

The previous two methods composed complete interactions. Another way to structure interactive programs is by the abstract device definitions of the Clean event I/O system. Because abstract device definitions are algebraic types, it is possible to define functions that create abstract device definitions that can be parameterized. The program in Figure 2.18 illustrates this idea. The function `simpleDraw` creates a very simple drawing window. It is parameterized with an id and a picture range. The interesting aspect of this window is that it can be applied to any interaction because it is polymorphic in its program state. Access to the program state is provided by further parameterization of three functions with type definitions :: `Add s` :== `Point s` → `s` to add a point to the program state, :: `Del s` :== `Point s` → `s` to delete a point from the program state, and :: `Get s` :== `s` → (`s`, `[Point]`) to retrieve all points drawn sofar. For instance, if the program state is [Point], then examples of the three operations are:

```
add :: Point [Point] -> [Point]
add p ps = [p:ps]

del :: Point [Point] -> [Point]
del p [q:qs]
|   p==q = del p qs
         = [q:del p qs]
del p [] = []

get :: [Point] -> ([Point], [Point])
get ps = (ps,ps)
```

The drawing functions `DrawPoint` and `ErasePoint` are both of type `Point *Picture` → `*Picture`. The mouse event handler `track` erases a point if both the mouse button and the option modifier key are pressed, and draws a point if the mouse is pressed (regardless of modifier keys). Figure 2.19 shows the drawing window in action.

```
simpleDraw::Id PictureDomain (Add .s) (Del .s) (Get .s) -> WindowDef .s
simpleDraw id dom add del get
=   Window "Picture" dom []
      [WindowId      id,
       WindowMouse   Able (track id add del),
       WindowUpdate  (update get),
       WindowCursor  CrossCursor]

update:: (Get .s) UpdateArea .s -> (.s,[*Picture -> *Picture])
update get _ s = (s1, map DrawPoint drawnPoints)
where
                 (s1, drawnPoints) = get s

track :: Id (Add .s) (Del .s) MouseState .s *(IOState .s)
      -> (.s, *IOState .s)
track _  _ _   (_,ButtonUp,_)        s io = (s, io)
track id _ del (point,_,OptionOnly) s io
=   (del point s, DrawInWindow id [ErasePoint point] io)
track id add _ (point,_,_) s io
=   (add point s, DrawInWindow id [DrawPoint  point] io)
```

Figure 2.18 A window definition for a very simple drawing program.

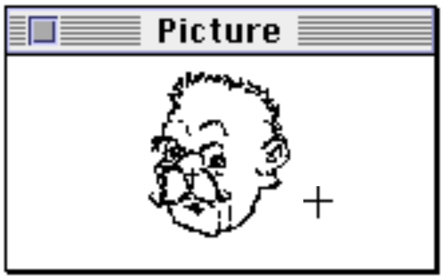


Figure 2.19 The drawing window in action.

2.8 Implementation of the interface

The Clean event I/O system is given structure by the abstract device concept. The programmer knows how to define abstract devices, and how to change them at run-time using abstract device access functions. The abstract device concept also gives structure to the implementation of the interface between the Clean event I/O system and concrete operating systems.

The interface between abstract devices and concrete devices boils down to five distinct actions for each abstract device: (**a**) hide concrete interface elements, (**b**) map abstract device definitions to concrete interface elements, (**c**) translate concrete events into abstract events and evaluate corresponding abstract event handlers, (**d**) free the resources of concrete interface elements, and (**e**) undo the hidden state of concrete interface elements. Each of these particular actions is a particular function, and so the interface between abstract and concrete devices is a structure of five *abstract device interface functions* (see Figure 2.20). The Boolean result of (**c**) indicates whether the abstract device interface function successfully evaluated the event.

```
:: DeviceFunctions s :== (HideFunction  s,      (a)
                          OpenFunction  s,      (b)
                          DoIOFunction  s,      (c)
                          CloseFunction s,      (d)
                          ShowFunction  s   )   (e)

:: HideFunction  s :==                    (IOState s)->IOState s
:: OpenFunction  s :== (DeviceSystem s)  (IOState s)->IOState s
:: DoIOFunction  s :== Event s            (IOState s)->(Bool,s,IOState s)
:: CloseFunction s :==                    (IOState s)->IOState s
:: ShowFunction  s :==                    (IOState s)->IOState s
```

Figure 2.20 The types of the abstract device interface functions.

The abstract device interface functions and the abstract device access functions provide the Clean event I/O system with an abstract view of the operating system for each abstract device. An important advantage of this approach is that porting the Clean event I/O system to other operating systems requires only the porting of these functions. Another advantage is that each operating system interface can exploit the underlying operating system in order to obtain efficient implementations.

2.9 Related work

As briefly discussed in the introduction, many solutions to deal with I/O in functional languages exist. In this section we discuss three approaches in detail: dialogue combinators (Dwelly, 1989), monads (Peyton Jones and Wadler, 1993), and FUDGETS (Carlsson and Hallgren, 1993).

2.9.1 Dialogue combinators

One of the early reports on how functional languages can be used to program dynamic and complex graphical user I/O is the paper on *dialogue combinators* by Dwelly (1989). Dialogue combinators are a class of functions with well-defined properties such that programs constructed by these functions behave in a predictable way. It is a discipline because programmers are not forced to program in this style. The type of a dialogue combinator is `Dlg s` = `s` → `[Inputs]` → (`[Outputs]`, `s`, `[Inputs]`). A dialogue combinator when applied to some object that represents the state of the program (the program state in our terminology) of type `s`, and a stream of user input of type `[Inputs]`, produces a triple consisting of some output of type `[Outputs]`, the new program state, and the user input that has not been consumed.

Programs are constructed by dialogue combinators, such as `NullDialogue`, `Join`, and `Cond`. `NullDialogue`: `Dlg s` produces no output, and leaves the program state and user input unchanged. The application of `Join`: `Dlg s` → `Dlg s` → `Dlg s` to two dialogue combinators `d1` and `d2` produces the dialogue combinator that first evaluates `d1` and then `d2`, and concatenates the output of `d2` to the output of `d1`. *Conditions* are functions of type `Cnd s` = `s` → `[Inputs]` → `Boolean`, which inspect the program state and the user input and yield a Boolean result. The application of `Cond`: `Cnd s` → `Dlg s` → `Dlg s` → `Dlg s` to a condition `c` and two alternative dialogue combinators `d1` and `d2` produces the dialogue combinator that performs `d1` if `c` holds and `d2` if not.

In order to program dynamic interfaces one special dialogue combinator, *`TreeCase`*, is provided. The basic idea behind this combinator is that dynamic interfaces can be defined by sets of *condition-action* pairs $[(c_1,a_1)\ldots(c_n,a_n)]$ or *rules*. The `TreeCase` combinator searches the first condition c_i of a rule i that is satisfied, and then applies the action a_i. The action is provided with the program state and the user input as usual, but also with the set of all current rules $[(c_1,a_1)\ldots(c_n,a_n)]$. The action may produce some output and change the program state as usual, but it also yields a new set of rules $[(c'_1,a'_1)\ldots(c'_m,a'_m)]$ which is recursively applied to `TreeCase`. A rule has type `Object tag s` = `Obj tag` (`Cnd s`) (`[Object tag s]` → `Dlg s`), and `TreeCase` is a function of type `[Object tag s]` → `Dlg s`. By providing an initial set of rules, the behaviour of `TreeCase` is determined if the user inputs are known.

Even though the dialogue combinator approach is stream based, our approach has remarkable similarities, as well as remarkable differences. The concepts on which the dialogue combinator approach is based, namely those of dialogue combinators as program state transition functions, sets of changing rules to program the behaviour of dynamic interfaces, and the `TreeCase` dialogue combinator to evaluate a dynamic interface can be retraced in our device concept. The main differences of the Clean approach are the elimination of event stream handling, the formalisation of the behaviour of the graphical user interface by the devices, and the modularisation of programs by allowing an arbitrary amount of interactions.

2.9.2 Monads

One of the currently widely investigated approaches to incorporate I/O in functional languages is the approach by Peyton Jones and Wadler (1993) based on *monads* (Moggi, 1989; Wadler, 1990-*b*). Support for the IO monad was originally implemented in the Glasgow Haskell compiler. A revised version of the specification is included in the definition of the latest version of Haskell (Hudak *et al.*, 1992). As we said in the introduction, this is an (implicit) environment based approach. The environment basically models the state of the machine. Operations (or *actions*) on the environment have a special type `IO a` which denotes "actions that, *when performed*, may do some I/O and then return a value of type `a`". For example, the actions `getcIO` :: `IO Char`, and `putcIO` :: `Char` → `IO` () read a character from standard input and write a character to standard output. The type () is a special type whose only proper element is the empty tuple, also written as ().

A programmer *composes* actions with two combinators `unitIO` :: `a` → `IO a`, and `bindIO` :: `IO a` → (`a` → `IO b`) → `IO b`. The application `unitIO x` denotes the action that only returns `x` and does no input nor output. The application `bindIO m n` (or using the Haskell infix notation ``m `bindIO` n``) for an action `m` :: `IO a`, and a function `n` :: `a` → `IO b`, first does `m`, which yields a value `x` of type `a`, and then does `n x`, which yields a value `y` of type `b`. These two combinators actually form the monad. Two other combinators are derived from `unitIO` and `bindIO`, namely `doneIO` :: `IO` () and `seqIO` :: `IO a` → `IO b` → `IO b`. The combinator `done-IO` simply does nothing. The application ``m `seqIO` n`` to two actions `m` :: `IO a` and `n` :: `IO b` first does `m` and then does `n`, and yields the result of `n`.

If we compare the monad approach to our approach then there are some striking differences. The environment that is manipulated in the monad approach is *implicit* and 'appears' only in the `IO` type. As a result, programming in the system creates one single spine of I/O operations and therefore over determines order of evaluation (see the discussion in Section 4.2), and combinators need to be provided in order to compose actions. To our knowledge it is not possible to define general combinators that combine arbitrary monads, but, using constructor classes, it is possible to combine specific monads with arbitrary monads (see Jones, 1993; 1995). It is interesting to look at two extensions to the `IO` monad that are used to create additional spines of I/O operations but in an unsafe manner, and see how these can be defined in the explicit environment style.

The first extension is the combinator `delayIO` :: `IO a` → `IO a` by which a program can *spark* an action that is evaluated *interleaved* with the main imperative spine. This is a dangerous combinator because the result of the program may depend on the evaluation order between the interleaved action and the main spine. It should only be used if the programmer proves that the interleaved action cannot interfere with the spine. In the explicit multiple environment passing style interleaved I/O is obtained because two spines of I/O can be defined on independent environments. There is no need for proof obligation because environments are independent by definition.

The second extension is the combinator `performIO` :: `IO a` → `a` by which a program sparks an action that is not connected to the main spine at all. Again, the programmer has to prove that the action cannot cause any side-effects in the program. This is clearly an example of unhygienic programming (end of Section 4.2). In the Clean event I/O system this situation cannot occur because an interactive program needs to yield a result value of type `*World`, which can only be done by applying the composition rule to the event stream and file system environments.

2.9.3 FUDGETS

The *FUDGETS* system by Carlsson and Hallgren (1993) is a system developed recently in which the stream based functional I/O approach is unified with graphical user interfaces (in particular, the X Windows system). The basic concept of the approach is the *fudget* (*fu*nctional wi*dget*). The fudget is the basic instrument to receive and handle concrete events, and send commands to other fudgets. Events and commands are communicated using streams. A fudget that accepts high level events of type α, and that sends commands of type β, is a fudget of type `F` α β. The system provides a set of primitive fudgets.

A program can create complex graphic interfaces by composing fudgets, and as usual a number of combinators are provided to do so. For instance, the combinator `>+<` puts two fudgets f_1 and f_2 of type `F` α_1 β_1 and `F` α_2 β_2, into a new fudget f of type `F` $(\alpha_1+\alpha_2)$ $(\beta_1+\beta_2)$. The notation $\alpha+\beta$ is shorthand for the algebraic type `Either` α β with `Either` α β = **`Left`** α | **`Right`** β. The new fudget f is the parallel composition of f_1 and f_2. Any message of type α_i is sent to f_i, which results in a response of type β_i. Program code is connected in a fudget structure by defining code as *stream processing functions* of predefined abstract data type `SP` α β. The operator `absF` turns such a function of type `SP` α β into an *abstract fudget* of type `F` α β. Because `SP` is an abstract data type combinators are provided to create stream processing functions, namely the *input* combinator `getSP` :: $(\alpha \rightarrow$ `SP` α $\beta) \rightarrow$ `SP` α β, and the *output* combinator `putSP` :: $[\alpha] \rightarrow$ (`SP` α $\beta) \rightarrow$ `SP` α β. The application `getSP (\a -> sf)` gets an incoming message of type `a` and continues as the stream processor `sf`. The application `putSP l sf` outputs the messages in `l` and continues as the stream processor `sf`. Fudgets can be created and destroyed dynamically using the `dynListF` combinator. Finally, to get an executable program, the fudget structure is offered to a function which takes care of the stream handling with the operating system.

In contrast with the Clean event I/O system, the FUDGETS system has no concept of a state that is accessible for the graphical user interface elements that are part of the interaction. All state is local to a fudget. For both event handling and communication, the FUDGETS system relies entirely on stream processing. Abstract fudgets are demanded to be written as stream processors, forcing a continuation style on the program.

2.10 Conclusions

In this chapter we have shown how file I/O and graphical user I/O have been incorporated in the lazy functional programming language Clean. File I/O is defined entirely using the explicit multiple environment passing method. This method allows explicit handling of resources from the outside world, such as files, event streams, windows, and so on. The direct use of the resources is safe due to the uniqueness types of Clean's typing system. In our experience, the restrictions that are imposed by the uniqueness type system to the programmer do not seriously hamper the functional expressiveness of the language. By the environment hierarchy the outside world is given structure, and multiple environments can be used in the same program independently. The complexity of programming graphical user interfaces is managed by introducing several stages of abstraction. A program is structured by partitioning it into a number of independent interactions. Each interaction can be considered on its own. An interaction is a dynamic state transition system which is constructed in a declarative style. An interaction is defined by an initial set of devices and an initial program state. The devices are defined by algebraic types which provide a concise and clear notation of the interface elements. The use of algebraic types to specify abstract devices and interactions is a very flexible tool, as algebraic types can be manipulated easily in a functional language.

The Clean event I/O library provides portability to very different operating systems. We have made implementations of the I/O library for the Sun under X Window system, using the Open Look Interface Tool kit, and the Macintosh. We are currently working on an implementation on PC's under OS/2. This means that a Clean application created and tested on a Macintosh only needs to be recompiled to run exactly the same on a X Window system. Still, the resulting applications obey the different look-and-feels of these systems. The library has been used to write several large applications (a full-feathered text editor, a relational database application, a spreadsheet, a Turing machine programming environment, and many games). The runtime performances of these programs are competitive with imperative programs.

Finally, we hope to have shown not only by the extensive account of the Clean I/O system (and in particular the uniqueness typing) but also by the related work, that functional languages have very strong organisational, abstractive, and expressive power. It is important that an I/O system for a functional language retains these strengths.

2.11 Current and future work

Research on the Clean I/O system as presented in this chapter has concentrated mainly on how to make I/O resources explicitly and safely available, and how to program graphical user I/O in a high-level and portable way. A technical report on the operational semantics of the Clean I/O system is in preparation (which will discuss the meaning of non-terminating interactive programs, and how to reason about interactive programs). Part of our current and future research focuses on making the I/O system more orthogonal. Concrete topics in this area are to what extent the window and dialogue device can be unified, and the completion of the set of functions to structure and combine interactive programs. The other main part of our research activities will be to investigate how the explicit multiple environment passing scheme can form a base for distributed (or parallel) interactive programs. Topics in this area are the environment hierarchy of a world that contains many other worlds, and the investigation of communication primitives between interactions.

Acknowledgements

We would like to thank Marko van Eekelen for comments on the text, and also for pointing out that hygienic use of environments is not a programming convention but rather an obligatory property of multiple environment passing schemes. We would also like to thank the referees of the paper for their valuable comments.