The Beauty and the Beast

Peter Achten, Rinus Plasmeijer
December 1992
University of Nijmegen, The Netherlands
peter88@cs.kun.nl, rinus@cs.kun.nl

Abstract

This paper presents a method for programming window-based I/O in a pure functional language. This method enables the programmer to specify interactions at a very high level of abstraction. It employs a machine independent declarative specification of devices such as windows, dialogues and menus. These devices are specified via predefined algebraic data types containing event handlers (user defined higher-order functions). All low level event handling is hidden from the programmer. The event stream is handled by a library function that takes the algebraic specification of the devices and applies recursively the appropriate event handler for each event.

The system is implemented for the lazy functional language Concurrent Clean. It has been used successfully for the implementation of large interactive applications such as a window based editor and a relational database. Due to the use of uniqueness types (that make it possible to define functions that perform updates without violating the functional semantics) applications run as efficient as their imperative counterparts: they can be actually used in practice.

We feel that the presented method turns I/O handling into one of the strengths of functional languages.

1. Introduction

Although we all love the beautiful aspects of functional languages we must admit that it is difficult to deal with a beast called Input-Output (I/O). If one wants to do I/O in a functional language, the need to do assignments is inevitable. Assignments have been banned from functional programming because of their unpleasant properties. However, a programming language without the full range of I/O facilities is not complete: the beauty cannot do without the beast. We have stalemate.

The language Concurrent Clean offers a linear type system called Unique Types which permits programs to define functions that perform destructive updates within a pure functional framework [11]. Unique Types offer a very broad and useful range of applications of which I/O is only one aspect. Using Unique Types makes it possible to do seeks in files or to write to a window in a very efficient way.

Although Unique Types allow us to do I/O while retaining a completely functional framework, programming I/O remains a very complicated task. In this paper we will mainly focus on window based I/O (Event I/O). In Event I/O the objects that are being manipulated are graphical interface objects such as windows, menus and dialogues. Graphical interface systems are event driven: the user of a program communicates with that program via interface objects. These actions of the user generate *events* to the program. The operating system also uses events to communicate to the program that things have been changed. Finally, manipulations of the interface objects by the program may generate events as well. All these events are merged in one event stream that can be accessed in the functional program. We will refer to these events as *crude events*.

By using functions that perform updates we can incorporate an imperative style of programming in a pure functional language (as e.g. can be done with monads [7]). Primitive I/O may provide all programming flexibility to the programmer and cause not much trouble in implementation. However, the consequences are that interactive programs will be specified at a low level, force the program into an imperative style, which is prone to errors, not portable and difficult to reason about. So although the beast is then beaten, it certainly has not changed into a prince.

The method we have designed to do I/O enables the specification of I/O at a very high level of abstraction (higher than the level offered by packages like the X Window System). We have combined the possibilities offered by the Unique Types with the expressive power of functional languages. Care has been taken that the structure of the system is as obvious as possible. The system enables programs to do I/O independent of their platform such that programs are completely portable.

We were mainly interested in the following aspects. What level of I/O specification should we offer the application programmer? How can we make specifications device independent yet suited for all kinds of platforms? Will large programs run efficient enough to be used in practice? Are the restrictions imposed by Unique Types a problem when large programs are written? Can one obtain a system that is flexible and extendible enough?

This paper is organised as follows. We start with a brief explanation of Concurrent Clean in section 2, followed by one about Unique Types and how they are used for destructive updates in section 3. Section 4 briefly discusses the explicit environment passing style and the environments that are used in Concurrent Clean. The basic building blocks to define interactive programs, the devices, are presented in section 5. They form the basis of every event driven program. Section 6 explains what interactions are and how they can be specified by means of devices. In section 7 we open black boxes and show how, hidden from the programmer, devices cooperate to get interactions going in a purely functional framework. Section 8 presents a number of ways in which existing interactions can be used to compose more complex ones by virtue of the framework itself and using common functional programming techniques. Some experience with large interactive applications is briefly discussed in section 9. Section 10 discusses a number of encountered problems and suggests some

ideas to cope with them. Finally the paper is concluded by section 11 which summarises the whole and presents some future work.

2. Concurrent Clean

Concurrent Clean [3],[6],[9] is a lazy functional programming language based on Term Graph Rewriting [2]. Here is an example of a Clean function defining the well-known fibonacci function.

```
::  Fib INT -> INT;       == type definitions start with ::
    Fib 1    -> 1;
    Fib 2    -> 1;
    Fib n    -> + (Fib (- n 1)) (Fib (- n 2)),   IF  > n 2
             -> ABORT "fibonacci: argument less than one";
```

Term Graph Rewriting systems are very suited for efficient implementations of functional languages [5],[9],[10]. 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 [9] and [4].

3. Unique Types and Destructive Updates

In this section we explain briefly the idea behind Unique Types in Clean and how it can be used to define functions that can do destructive updates. For more information about Unique Types we refer to [11] and [9].

Assume that we want to define a function FWriteC of type :: CHAR FILE -> FILE that upon evaluation directly writes a character to the given file. It is the obvious way to do it in any imperative language but it is not sound to do this in the functional world 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 one function can therefore also affect the outcome of other computations that share the same argument. This is illustrated in the following example:

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

Now, when both files are shared in the function body of F writing the character 'a' to the file will produce the wrong result (file++"a", file++"a"). To conform with the standard semantics FWriteC has to yield a new file and the result then correctly becomes (file, file++"a"). But, constructing new files is of course very inefficient and it is not the intention either. One really wants to have the ability to modify (update) an *existing* file *instantaneously*.

Fortunately, updates *can* be allowed under certain conditions. *If* it can be guaranteed that an offered argument is not used by (shared with) other function applications it can be re-used for the construction of the function result e.g. via a destructive update. In Concurrent Clean, a type system is incorporated [4], [9] that guarantees that certain objects (unique objects) can be reused safely. The UNQ type attribute can be added by the programmer to *any* type to express the restricted use of an object of that type. For instance, when the type of the function FWriteC is specified as :: CHAR UNQ FILE -> UNQ FILE it is guaranteed that any application of the function FWriteC is called with an argument of type FILE that is not used somewhere else. So, the dangerous example given above is not possible because it is not approved by the type system. The UNQ type of the resulting file indicates that no sharing is introduced in the constructed result such that the resulting modified file can be passed to another call of FWriteC. In this way single threaded use of objects is enforced.

4. The I/O World of Concurrent Clean

The example of writing a character to a file is quite illustrative for the approach we have taken to low-level I/O in general in Concurrent Clean. I/O is always done on objects (like the file in the example). Such an I/O object serves as an environment. Environments are offered to the application programmer as abstract data types. Functions defined on these structures that change the environments all require the environments to be *unique*, so they can indeed be updated destructively without harm. Passing environments around only to those functions (predefined and program defined) that actually need them is called the “explicit environment passing where needed” scheme.

Concurrent Clean provides four types of predefined environments to perform I/O. These are WORLD, FILES, FILE and IOState. The WORLD is the main container of all other environments. There is at most one WORLD for each program. It can be retrieved only at the initial Start rule of a program. From this WORLD disjoint sub worlds can be retrieved. One such sub world is an object of type FILES, an abstract data structure representing the state of the file system. It gives access to all files (of type FILE) visible for the program. A FILE can be opened read-only or writable in which case it is an UNQ FILE. Another disjoint sub world environment, the IOState, is used by programs to do event I/O. It can be retrieved only from the WORLD and contains among other things the event stream.

Using a hierarchy of environments from which others can be retrieved has a number of important properties and advantages above traditional environment passing methods in which there exists only one monolithic environment. Our system allows a natural way to handle *multiple states*. The most obvious example of multiple state handling is the use of individual files in different parts of the program. The evaluation order of the program is fixed only by the order of function applications on the environment. This increases the clarity of the program. It is our belief that multiple state handling is essential to achieve distributed I/O.

5. Devices

To *users* of graphical interface environments like Macintosh or X Window System, all programs communicate with them by means of dialogues, windows and menus. All information and actions are directed to these interface elements. Users get familiar with applications by their interface elements, and there is no need to be bothered by whatever system is underneath.

To *programmers* of graphical interfaces, the *same interface elements* are provided by the Concurrent Clean Event I/O library by which the programmer *constructs* interactive programs. These elements, the *devices*, are defined on a level very close to the way they would appear to him if he were a user of the program.

Concurrent Clean has four predefined devices: the MenuDevice, DialogDevice, WindowDevice and TimerDevice. Programs specify devices obeying a predefined *algebraic type*. For each device there is a predefined algebraic type: MenuDef, DialogDef, WindowDef and TimerDef. The algebraic type completely specifies the device: what it should look like and what event handlers should be called in what situations. Event handlers are *program defined functions* that specify the response of the program.

The remainder of this section discusses each of the devices. Readers who wish to know how devices are used to build interactive programs may skip the remainder of this section and go on to section 6 and 7.

5.1. MenuDevice

The MenuDevice conceptualises choosing commands from a set of available commands. The set one can choose from is rather simple. Figure 1 shows the predefined MenuDef algebraic type as it is offered to the Clean programmer. It specifies a simple language in which arbitrary menus can be defined.

```
TYPE
::  MenuDef UNQ s UNQ io
    -> PullDownMenu MenuId MenuTitle SelectState
           [MenuElement s io];
::  MenuElement UNQ s UNQ io
    -> MenuItem MenuItemId ItemTitle KeyShortcut
           SelectState (MenuFunction s io)
    -> CheckMenuItem MenuItemId ItemTitle KeyShortcut
           SelectState MarkState (MenuFunction s io)
    -> SubMenuItem MenuId ItemTitle SelectState
           [MenuElement s io]
    -> MenuItemGroup MenuItemGroupId [MenuElement s io]
    -> MenuRadioItems MenuItemId [RadioElement s io]
    -> MenuSeparator;
::  RadioElement UNQ s UNQ io
    -> MenuRadioItem MenuItemId ItemTitle KeyShortcut
           SelectState (MenuFunction s io);
::  MenuFunction UNQ s UNQ io  -> => s (=> io (s, io));
::  MenuTitle                  -> STRING;
::  ItemTitle                  -> STRING;
::  MenuId                     -> INT;
::  MenuItemId                 -> INT;
::  MenuItemGroupId            -> INT;
::  KeyShortcut    -> Key KeyCode | NoKey;
::  KeyCode        -> CHAR;
::  SelectState    -> Able | Unable;
::  MarkState      -> Mark | NoMark;
```

Figure 1 *The algebraic type MenuDef to define menus. The symbols in bold are data constructors, the symbols in italics are user definable functions that will be called to handle the corresponding menu event. The other symbols are type symbols. A menu is a pull down menu which contains a number of menu elements. Sub menus, item groups and radio items are elements that contain other elements. (Check) menu items are the elements that actually can be selected. Elements can always be selected by mouse. An item can be selected with the keyboard by means of KeyShortcut. The menu function of an item is the program defined function (the event handler) that is called when the item is selected. Items as well as menus can be disabled or enabled.*

All parts of a menu have an identification attribute (The synonym types MenuId, MenuItemGroupId and MenuItemId). At run-time the program can change the attributes of elements by referring to them by their identification.

The layout of a menu strongly corresponds with the data constructors that are used in its definition. To define a menu a programmer actually groups the commands he wishes the system to have and their initial looks. Example 1 shows the definition of a concrete menu. Its appearance on a Macintosh system is shown in figure 2.

```
RULE
::  MyMenu -> MenuDef State (IOState State);
    MyMenu
    -> PullDownMenu FileId "File" Able
           [MenuItem SetFigureId "Set Figure..." (Key 'F')
               Able SetFigure,
            MenuSeparator,
            MenuItem QuitId "Quit" (Key 'Q') Able Quit ];

MACRO FileId -> 1;    SetFigureId -> 10;  QuitId -> 11;
```

Example 1 *A small pull down menu. The last line defines a number of macro definitions. A macro is a rewrite rule that is rewritten at compile-time: all occurrences of the left-hand side of a macro definition are substituted by the right-hand side. Macros may be parameterised.*

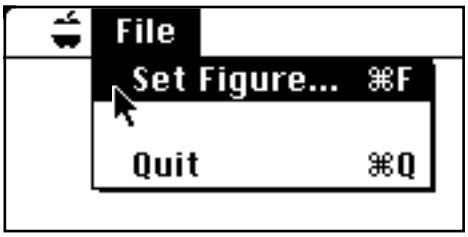


Figure 2 *The menu system of example 1 as it appears on a Macintosh system. Note the close correspondence between the specification and the appearance on the screen.*

5.2. DialogDevice

The DialogDevice models structured communication between program and user. Applications can have an arbitrary number of dialogues open. The DialogDevice manages property and command dialogues, as well as notices. Dialogues may require a special mode of event handling: modal dialogues enforce users to deal with the dialogue entirely before any other actions can take place. Modeless dialogues are less demanding: even though they are open, the user may disregard them and use them when convenient. Property dialogues are always modeless, and are used to set properties of the interaction. Command dialogues can be modal or modeless. Property and command dialogues can contain editable,

static and dynamic texts, radio buttons, check boxes, pop up items, buttons (standard or program defined) and program defined controls. Notices are very simple modal dialogues to inform the user about unusual or dangerous situations.

So, the definition of a dialogue is a lot more complicated than menu definitions. This is not only due to the larger number of different kinds of items possible in a dialogue, but also because the *layout* between the items has to be defined. The DialogDef algebraic type defines a simple layout system that is sufficiently powerful to define quite complex dialogues. Like menus, the items of a dialogue are summarised in a list. Each dialogue item has a layout attribute of type ItemPos (see figure 3). The default layout strategy is to place items from top to bottom, from left to right.

```
TYPE
::   ItemPos -> Left | Center | Right |
                RightTo DialogItemId |
                Below DialogItemId |
                XOffset DialogItemId Measure |
                YOffset DialogItemId Measure |
                XY Measure Measure |
                ItemBox INT INT INT INT;
::   Measure -> MM REAL | Inch REAL | Pixel INT;
```

Figure 3 *The algebraic type ItemPos by which dialogue items can arrange their layout. The constructors Left, Center and Right align the item to the left, centre it or align it to the right. RightTo places the item to the right of the item with the given id. If that item happens to be centred then the two of them are centred. Below places the item below the item with the given id. XOffset and YOffset place the item exactly some distance from the item with the given id. XY places an item on a precise location, and ItemBox also defines what part is visible (in pixel co-ordinates).*

Example 2 shows the definition of a concrete dialogue commonly found in text editors or word processors.

```
RULE
::   MyDialog -> DialogDef State (IOState State);
     MyDialog
     -> CommandDialog FindId "Find" Modeless
          [DialogSize (MM 85.0) (MM 50.0)]
          FindButtonId
          [StaticText FindTextId Left "Find:",
           EditText FindStringId
              (YOffset FindTextId (MM 1.0)) (MM 70.0) 1 "",
           CheckBoxes SettingsId Left (Rows 2)
              [CheckBox IgnoreCaseId "Ignore Case" Able Mark F,
              CheckBox BackwardId "Backward" Able NoMark F,
              CheckBox WrapRoundId "Wrap Around" Able Mark F,
              CheckBox MatchWordsId "Match Words" Able Mark F
              ],
           DialogButton CancelId Left "Cancel" Able Cancel,
           DialogButton FindButtonId (RightTo CancelId)
                 "Find" Able Find ];
```

Example 2 *A command dialogue to find texts as one might find in an editor. It is titled Find and is modeless. It's size (width and height) is fixed by the DialogSize attribute. The default button is the one with id FindButtonId, which is the button named Find. The dialogue contains an editable text field in which the user can type the text to be found. There are four check boxes, arranged in two columns, by which the user can influence the way searching will be done. The Find button will search for the text. It can be selected by mouse, or because it is the default button, by pressing the return key (depending on the feel of the system).*

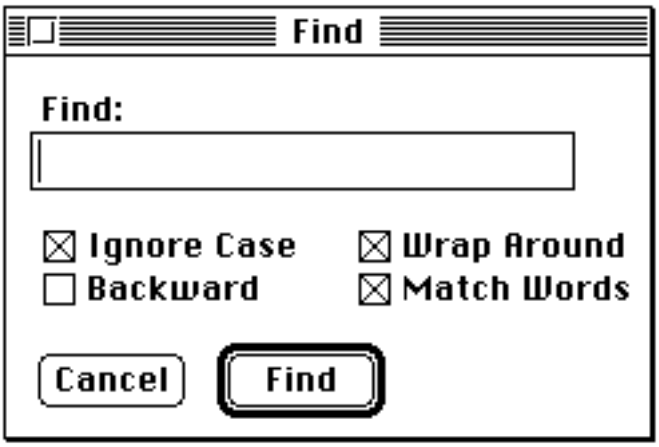


Figure 4 *The dialogue specified in example 2 as it will appear initially on a Macintosh system.*

The Concurrent Clean library provides a large number of predefined dialogue items that are common on all systems. Using these items in a dialogue guarantees that the dialogue will look and feel like any other dialogue of the system. For programmers who need to have more sophisticated or very specialised dialogue items Controls have been provided which are dialogue items that can be fully defined by the programmer. With such items a programmer can define his own slider bars or scrolling lists. Figure 5 shows the algebraic type definition. The programmer in fact defines a look-and-feel of his own which is system independent. So the use and looks of Controls are exactly the same for all platforms.

```
TYPE
::   DialogItem UNQ s UNQ io
     -> ...
     -> Control DialogItemId ItemPos PictureDomain SelectState
              ControlState ControlLook ControlFeel
              (DialogFunction s io);

::   ControlState -> BoolCS BOOL |
                     IntCS INT |
                     RealCS REAL |
                     StringCS STRING |
                     CS [ControlState];
::   ControlLook
     -> => SelectState (=> ControlState [DrawFunction]);
::   ControlFeel
     -> => MouseState  (=> ControlState
                           (ControlState,[DrawFunction]));
::   DrawFunction -> => Picture Picture;
::   DialogFunction UNQ s UNQ io
     -> => (DialogDef s io) (=> (DialogState s io) (DialogState s io));
```

Figure 5 *The algebraic type to make user defined dialogue items. Like any other dialogue item controls have a dialogue item id and are positioned in the dialogue. The PictureDomain defines a co-ordinate system for the control in which its look is defined and that is used when mouse events are passed. The SelectState determines whether the control is selectable or not. The essential part of a control definition is given by the functions ControlLook and ControlFeel. ControlLook defines what the control should look like depending on the*

(un)selectability and current control state. ControlFeel defines how the control responds to user actions by mouse and the control state. User actions may change the control state. For the control state the programmer can use one of the basic types boolean, integer, real, string or a list of them; whichever happens to be most convenient. Finally, the DialogFunction of the control defines the influence of the control on other items of the dialogue.

Suppose we are not satisfied with the check boxes in the find dialogue in figure 4 and want to define our own check boxes. Example 3 shows the Concurrent Clean definition

```
RULE
::  MyCheckBox DialogItemId ItemPos SelectState MarkState
        (DialogFunction UNQ s UNQ io)
        -> DialogItem UNQ s UNQ io;
    MyCheckBox item_id pos ability markstate function
    ->  Control item_id pos MarkBox ability
            (BoolCS (Marked markstate)) Look Feel function;

::  Look SelectState ControlState -> [DrawFunction];
    Look ability (BoolCS marked)
    ->  [interior, backslash, slash | box]      IF able && marked
    ->  [interior | box]                        IF able:Enabled ability
    ->  [backslash, slash | box]                IF marked
    ->  box,
        interior    : DrawRectangle
                        ((1,1),(-- MarkBoxWidth, -- MarkBoxHeight)),
        backslash   : DrawLine ((2,2), (width, height)),
        slash       : DrawLine ((width, 1), (2, -- height )),
        box         : [DrawRectangle MarkBox,
                        DrawRectangle ((2,2),(width, height))],
        width       : - MarkBoxWidth 2,
        height      : - MarkBoxHeight 2;

::  Feel MouseState ControlState -> (ControlState,[DrawFunction]);
    Feel (pos, ButtonUp, modifiers) (BoolCS marked)
    ->  (new_state, [EraseRectangle MarkBox | look]),
        new_state   : BoolCS (NOT marked),
        look        : Look Able new_state;
    Feel mouse_state control_state -> (control_state, []);

::  Enabled SelectState -> BOOL;
    Enabled Able -> TRUE;
    Enabled Unable -> FALSE;

::  Marked MarkState -> BOOL;
    Marked Mark -> TRUE;
    Marked NoMark -> FALSE;

MACRO
    MarkBox         -> ((0,0),(MarkBoxWidth, MarkBoxHeight));
    MarkBoxWidth    -> 18;
    MarkBoxHeight   -> 18;
```

Example 3 *A small example of a control definition, defining a variation on check boxes with a double border. Since we only need to administrate whether the check box is marked or not, we use the BoolCS control state which is TRUE when marked and FALSE if not. The PictureDomain of the check box has a fixed size, defined by MarkBox. The look of the check box is defined by Look: there are four variations possible depending on the selection state and control state (see figure 6). The feel of the check box is defined by Feel: when the user presses the mouse in the box, nothing happens. Only when the button is released, the control state changes by negating its boolean value.*

Figure 6 *MyCheckBox in the following conditions (from left to right): when not able and marked, not able and unmarked, able and marked and when able but unmarked.*

5.3. WindowDevice

Windows are the basic medium in which interactive applications communicate with users. An application can have an arbitrary number of windows open. Of these windows at most one is *active*. The active window is the window to which all keyboard events are directed. The application as well as the user can decide which window to make the active window. The active window is easy to identify to users. Window systems structure the direction of user input to applications.

Applications can display anything in a window: a window gives a view on a Picture. Pictures are abstract UNQ data types in the Concurrent Clean I/O library on which a very large number of drawing functions is defined. The I/O system provides two types of windows: scrollable windows (**ScrollWindow**) and fixed size windows (**FixedWindow**). The latter windows are the most simple ones: their content is always completely visible. Scrollable windows view pictures that are arbitrarily large. With the scroll bars the user changes the current view on the picture.

The most important program-defined window event handlers are the *update function* and the *mouse* and *keyboard handlers*. With the update function the programmer defines the content of (any part of) a window. The I/O system frequently uses the same function for scrolling, resizing windows, updating affected parts of the window and so on. The mouse and keyboard handlers define how the window responds to mouse and keyboard events. The I/O system supplies the mouse handler with information (the data type MouseState) about the position of the mouse, the status of mouse button(s) and modifier keys that were pressed simultaneously. The keyboard handler is provided with information (the data type KeyboardState) about the key involved, its status and also the modifier keys that were pressed simultaneously (see figure 7).

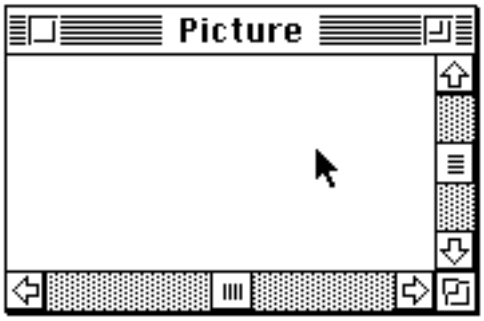
```
TYPE
::  UpdateFunction UNQ s ->
        => UpdateArea (=> s (s, [DrawFunction]));
::  UpdateArea -> [Rectangle];

::  MouseFunction UNQ s UNQ io ->
        => MouseState (=> s (=> io (s, io)));
::  MouseState -> (MousePosition, ButtonState, Modifiers);
::  MousePosition -> (INT, INT);
::  ButtonState -> ButtonUp | ButtonDown |
                   ButtonDoubleDown |
                   ButtonTripleDown |
                   ButtonStillDown;

::  KeyboardFunction UNQ s UNQ io ->
        => KeyboardState (=> s (=> io (s, io)));
::  KeyboardState -> (KeyCode, KeyState, Modifiers);
::  KeyCode -> CHAR;
::  KeyState -> KeyUp | KeyDown | KeyStillDown;

::  Modifiers -> (BOOL,BOOL,BOOL,BOOL);
```

Figure 7 *The three main functions for windows. For each modifier (Shift, Option, Command, Control) a Boolean in Modifiers indicates whether it was pressed (TRUE) or not (FALSE). On systems that have no Command key both the*

third and the fourth Boolean become TRUE when Control is pressed.

Example 4 shows a window definition and its appearance on a Macintosh system.

```
RULE
::  MyWindow -> WindowDef State (IOState State);
    MyWindow
    ->  ScrollWindow MyWindowId (0,0) "Picture"
            (ScrollBar (Thumb 450) (Scroll 10))
            (ScrollBar (Thumb 450) (Scroll 10))
            ((0,0),(1000,1000)) (50,50) (160,80) Update
            [Mouse Able HandleMouse,
             Keyboard Able HandleKeys ];
```



Example 4 *A window definition and its initial appearance on a Macintosh system. The window defines a view on a picture with a picture domain with origin (0,0) and a range of 1000 by 1000 pixels. The minimum size of the window is 50 by 50 pixels and its initial size 160 by 80 pixels. The initial view of the window on the picture is at the position (450, 450) which is determined by the values of the horizontal and vertical scroll bars. When the user scrolls through the picture, this is done in steps of 10 pixels. For this window, the thumbs will always be adjusted on a multiple of 10.*

5.4. TimerDevice

The TimerDevice enables interactions to synchronise on time intervals. The TimerDevice only responds to timer events; special events that are generated when the given time interval has expired. This mechanism cannot provide real-time timing because event handlers may take a longer evaluation time than the interval of the timer. The timer event handler is therefore provided with the number of discrete intervals that have passed.

```
TYPE
::  TimerDef UNQ s UNQ io
    ->  Timer TimerId SelectState TimerInterval (TimerFunction s
io);
::  TimerFunction UNQ s UNQ io
    -> => TimerState (=> s (=> io (s, io)));
::  TimerId         -> INT;
::  TimerInterval   -> INT;
::  TimerState      -> INT;
```

Figure 8 *The algebraic type TimerDef to define timers.*

```
RULE
::  MyTimer -> TimerDef State (IOState State);
    MyTimer -> Timer MyTimerId Able TicksPerSecond DoBeep;
::  DoBeep TimerState State (IOState State) ->
        (State, IOState State);
    DoBeep time_passed state io -> (state, Beep io);
```

Example 5 *A timer device which emits a beep every second. TicksPerSecond is a system constant which defines the number of ticks that take a second. DoBeep ignores the time that has actually elapsed.*

5.5. Summary

Algebraic types have proved very useful as device definitions. They provide a simple specification language that can be formally verified by the compiler's type checker. Careful choice of type names and constructors enable clear specifications and readable definitions. Both aspects help to prevent and detect the occurrence of many evident errors. Care has been taken to realise a one-to-one correspondence between specification and appearance (WYSIWYS: What You Say Is What You See).

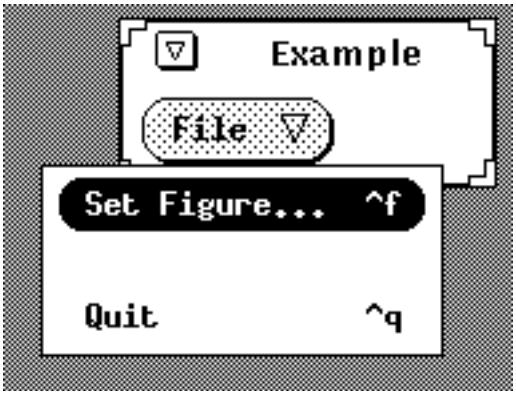


Figure 9 *The menu system of example 1 as it appears on X Window system, using the Open Look Interface Tool kit.*

Devices capture only the essence of the interface elements they model. For a programmer this yields a declarative and compact specification of the I/O he wants done. It enables implementations to abstract from concrete platforms, so devices can be made *look-and-feel independent*. This implies that a program developed on a Macintosh system will behave like a Macintosh application, but when recompiled (not rewritten!) on X Window System with the Open Look™ Interface Tool kit, it will behave like a standard Open Look application (see figure 9).

Programmers only specify devices: how they work and what they really are is hidden from them. Therefore the programmer need not reinvent the wheel of event handling over and over again in every program.

For each device Concurrent Clean provides a library of functions by which the program can change attributes of any of its parts. These are always referred to by their id's. With these functions devices can obtain a highly dynamic behaviour.

6. Interactions

Having seen how to define single devices, we can now start to create *interactions*. An interaction is the specification of a *state transition system* in which I/O is done. Be careful about the difference between interactions and programs! A program can have several interactions. A single interaction can be arbitrarily large and complex and will, in most cases, actually be the whole program.

State transition systems generally are fixed by an *initial state*, a set of all possible *transitions* and a *final state*. Here we transfer these concepts to interactions.

The *state* of every interaction is composed of two objects: the *program state* and the *I/O state*. The program state is a data structure of arbitrary UNQ type the interaction uses to keep all global data the interaction needs during evaluation. It is defined by the application programmer and depends on the specific application that is being defined. The I/O state is modelled by the sub environment IOState (introduced in section 4). The IOState is a container for the event stream and the devices that participate in the interaction.

The *initial state* of an interaction consists of the initial program state and the initial IOState. The initial program state is of course provided by the application program. An initial IOState is created by retrieving it first from the WORLD. This IOState contains the current event stream but has yet to be filled with device definitions. These devices are defined in the way as described in the previous section. These definitions are collected in a data structure IOSystem; a list of all device definitions. These devices operate on the same program state enforced by the type.

```
TYPE
::  IOSystem UNQ s UNQ io -> [DeviceSystem s io];
::  DeviceSystem UNQ s UNQ io
    -> TimerSystem    [TimerDef    s io]
    -> MenuSystem     [MenuDef     s io]
    -> WindowSystem   [WindowDef   s io]
    -> DialogSystem   [DialogDef   s io];
```

Figure 10 *The algebraic type IOSystem is used for defining a group of devices which will participate in the same interaction.*

The *transition set* of an interaction is the collection of all event handlers of the specified devices. The system automatically derives them from the participating devices. Transitions are triggered by the crude events in the event stream. The event is dispatched to the proper device which computes the next interaction state (with or without the use of a program event handler).

Interactions are started by the rule StartIO of type :: (IOSystem s) s (IOState t) -> (s, IOState t). In its arguments we can identify the state transition elements we just discussed. The initial interaction state is given by the argument s which gives the initial program state; the arguments IOSystem and IOState determine the respective initial devices and IOState. As mentioned, all event handlers can be derived from the participating devices; so we can now start to evaluate the interaction.

Interactions are handled recursively by StartIO until they reach their *final state* when one of the program event handlers *quits* the interaction. The interaction is quitted by applying the special rule QuitIO to the current IOState. QuitIO removes all devices from the IOState. The final interaction state is formed by the currently reached program state and the emptied IOState.

```
TYPE :: S -> [Point];

RULE
::  Start WORLD -> (S, WORLD);
    Start world -> (final, CloseIOState io' world'),
        (final, io'): StartIO [WindowSystem [Window]] [] io,
        (io, world'): OpenIOState world;

::  Window -> WindowDef S (IOState S);
    Window
    -> FixedWindow 1 (0,0) "Picture" ((0,0),(160,80))
        [Mouse Able Track,
         GoAway Quit;
         Cursor CrossCursor],

::  Track MouseState S (IOState S) -> (S, IOState S);
    Track (pos, ButtonUp, modifiers) state io -> (state, io);
    Track (pos, button_down, modifiers) state io
    ->  ([pos | state], DrawInActiveWindow [DrawPoint pos] io);

::  Quit S (IOState S) -> (S, IOState S);
    Quit state io -> (state, QuitIO io);
```

Example 6 *A window tracking the mouse. Window defines a FixedWindow with id 1, which should be opened with its upper left corner on screen co-ordinates (0,0), has the title Picture and a content 160 pixels wide and 80 pixels high. Track does nothing if the mouse button is up, otherwise it adds the current location of the mouse in the program state and draws the point in the active window.*

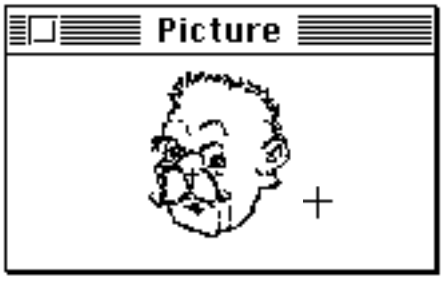


Figure 11 *Example 7 running.*

Example 6 gives a simple but *complete* drawing program, as one may write it in Concurrent Clean. The program state S consists of a list of points that are collected by tracking the mouse. The initial program state is of course the empty list [], since nothing has been drawn sofar. The initial IOState is retrieved from the WORLD. The initial set of devices is the WindowDevice only. One can see at a glance that it will also be the sole device of the interaction and that the only transitions that are possible are mouse events. The mouse events in the window are handled by the event handler *Track* and closing the window is handled by *Quit*.

7. The Soul of the Beast

In the previous sections we have assumed devices to be black boxes that in some way manage crude events. In this section we give a detailed description of the semantics of devices and how interactions, evaluated by StartIO and terminated by QuitIO actually work in a pure functional framework. All functions but StartIO and QuitIO discussed in this section are not visible to the application programmer. The whole mechanism of devices and evaluation of interactions is hidden from the programmer.

In order to describe the behaviour of the devices offered by the Clean I/O system, we can filter out five distinct actions defined for each device: (**a**) devices must be hidden, (**b**) device definitions must be mapped into the real world, (**c**) devices must determine which crude events they handle and handle them if they occur, (**d**) devices must close themselves and remove their mapped instances from the real world and finally, (**e**) devices must reappear again from a hidden state. Each of these particular actions is a particular

function, and so a device is defined by a set of five functions, each defining a specific part of a device behaviour (see figure 12).

```
TYPE
:: DeviceFunctions UNQ 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) (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 12 *The type of the functions defining the behaviour of a device.*

The OpenFunction (**b**) when applied to a device definition and the current IOState will create a concrete device according to the definition and store it in the IOState. The CloseFunction (**d**) when applied to the current IOState retrieves the concrete device from the IOState and frees all occupied system memory or resources.

The DoIOFunction (**c**) is applied to the current crude event, the current program state and IOState. It first checks whether the given event belongs to its input domain of events. If this is the case the Boolean result is TRUE; if not it is FALSE. Then the function determines whether it can manage the event all by itself or if it needs to call one of the program defined event handlers. Program defined event handlers may require not only the current program state and IOState but also some additional information (like the function Track of example 6 which is also applied to the MouseState).

The HideFunction (**a**) when applied to the current IOState hides the device for the user. The ShowFunction (**e**) when applied to the current IOState makes the hidden devices reappear on screen.

The functions StartIO and QuitIO that are responsible for doing interactions actually know nothing about devices. They only know the device functions of each device and can best be regarded as dispatchers: they only conduct events to devices.

StartIO actually consists of the same phases (**a**) to (**e**) each defining part of a device's behaviour. It starts to hide all present devices from the current IOState by subsequently applying the HideFunction of those devices on the resulting IOStates. Then all new devices are stored in the IOState by subsequently applying the OpenFunction of the new devices on the IOState with the corresponding device definitions. This is followed by retrieving crude events from the IOState and dispatching these events to the devices, by subsequently applying the DoIOFunction of each device. In case an event handler has Quitted the IOState the event loop is terminated. StartIO terminates after all hidden devices from the previous interaction have been shown again by subsequently applying those devices ShowFunction.

```
::  StartIO (IOSystem s (IOState s)) s (IOState t) -> (s, IOState t);
    StartIO device_definitions initial_program_state current_io_state
    ->  (final_program_state, ShowIO final_io_state),
        (final_program_state, final_io_state):
            DoIO initial_program_state (
                OpenIO device_definitions (
                    HideIO current_io_state));

::  HideIO (IOState t) -> IOState s;
    HideIO io_state -> HideDevices io_state Devices;

::  HideDevices (IOState t) [Device] -> IOState s;
    HideDevices io_state [device | devices]
    ->  HideDevices (hide io_state) devices,
        (show, open, do_io, close, hide): DeviceFunction device;
    HideDevices io_state [] -> io_state;

::  OpenIO (IOSystem s (IOState s)) (IOState t) -> IOState s;
    OpenIO [device_definition | device_definitions] io_state
    ->  OpenIO device_definitions (open device_definition io_state),
        (show, open, do_io, close, hide):
            DeviceFunction device_definition;
    OpenIO [] io_state -> io_state;

::  DoIO s (IOState s) -> (s, IOState s);
    DoIO current_program_state io_state
    ->  (program_state, next_io_state),      IF closed
    ->  DoIO program_state next_io_state,
        closed: IOStateIsClosed next_io_state,
        (program_state, next_io_state):
            DevicesDoIO event Devices current_program_state
                                                io_state',
        (event, io_state'): GetEvent io_state;

::  DevicesDoIO Event [Device] s (IOState s) -> (s, IOState s);
    DevicesDoIO event [device | devices] program_state io_state
    ->  (next_program_state, next_io_state),    IF mine
    ->  DevicesDoIO event devices next_program_state
                        next_io_state,
        (mine, next_program_state, next_io_state):
            do_io event program_state io_state,
        (show, open, do_io, close, hide): DeviceFunction device;
    DevicesDoIO event [] program_state io_state
    ->  (program_state, io_state);

::  ShowIO (IOState s) -> IOState t;
    ShowIO io_state -> ShowDevices io_state Devices;

::  ShowDevices (IOState s) [Device] -> IOState s;
    ShowDevices io_state [device | devices]
    ->  ShowDevices (show io_state) devices,
        (show, open, do_io, close, hide): DeviceFunction device;
    ShowDevices io_state [] -> io_state;

::  Devices -> [Device];
    Devices -> [TimerDevice,
                MenuDevice,
                WindowDevice,
                DialogDevice];
```

Figure 13 *The definition of StartIO.*

QuitIO simply calls each CloseFunction of the devices of the IOState to let them remove themselves from the IOState. The resulting IOState only contains the event stream.

```
::  QuitIO (IOState s) -> IOState s;
    QuitIO io_state -> CloseDevices io_state Devices;

::  CloseDevices (IOState s) [Device] -> IOState s;
    CloseDevices io_state [device | devices]
    ->  CloseDevices (close io_state) devices,
        (show, open, do_io, close, hide): DeviceFunction device;
    CloseDevices io_state [] -> io_state;
```

Figure 14 *The definition of QuitIO.*

All code presented in this section is pure Concurrent Clean and is in fact a fragment of the definition of the operational semantics of the Concurrent Clean Event I/O system [1].

8. Composing Interactions

With one call to StartIO a complete interaction is defined and handled entirely by the system. For most interactive applications this is already the whole program. However, it is even possible during an interaction to change completely from that I/O set-up to another (*nested I/O*), or to have a *sequence* of I/O set-ups.

Nested interactions are created simply by nested calls to StartIO. This can be done *anywhere inside an interaction*. Let interaction A have an event handler F of type :: s (IOState s) -> (s, IOState s) with the following definition:

```
::  F s (IOState s) -> (s, IOState s);
    F a io_a
    ->  (UseFinalStateB b a, io_a'),
        (b, io_a'): StartIO IOSystemB InitStateB io_a;
```

The rules IOSystemB and InitStateB define the new initial devices of interaction B and the initial program state respectively. B's program state can be of a completely different type than the program state of A. What happens is that when event handler F is called interaction A will be completely taken over by interaction B. B can have its own menu system, its own dialogues, windows and timers. While interaction B is the current interaction all devices of A are hidden from the user. Only when B terminates (because one of its event handlers quitted the interaction) are the devices of A shown again and the evaluation of interaction A is continued. Interactions can be nested arbitrarily deeply.

Sequences of interactions are also defined in a straightforward way. Let A and B be functions of type :: WORLD -> WORLD. Now it is easy to compose the function C of the same type:

```
::  C WORLD -> WORLD;
    C world -> B (A world);
```

C has the obvious meaning of evaluating function B after A. So if A and B are interactions, we have now composed a new interaction C which first handles interaction A and then B.

It may be the case that we want interaction B to depend on the final program state of interaction A. This can be achieved easily by combining the StartIO rules of both interactions:

```
::  C WORLD -> WORLD;
    C world ->  CloseIOState io_b world',
        (b, io_b): StartIO (IOSystemB a) (InitStateB a) io_a,
        (a, io_a): StartIO IOSystemA InitStateA io,
        (io, world'): OpenIOState world;
```

By changing the type of the interactions into :: (SA, WORLD) -> (SA, WORLD) and :: (SA, WORLD) -> (SB, WORLD) respectively we can have interaction B depend on the final program state of interaction A in a style more like the first sequence:

```
::  C (SA, WORLD) -> (SB, WORLD);
    C initStateA&World -> B (A initStateA&World);
```

These examples only indicate some ways to compose interactions. All the expressive power of functional programming can be used to build new interactions out of others.

9. Experience with Interactive Applications

In this section we discuss some of our programming experience and some of the applications we have written.

The structure of (both large *and* small) interactive applications can generally be based on the two major components of an interaction: the program state and the devices that will engage in the interaction. When writing a large application it is good practice to define the program state in a separate module and define access functions on all fields of the program state: one should treat the program state as if it were an abstract data type (even if it is not). The advantage is that when the program state changes, only the access functions need to be changed.

We have found that when designing the interface, it is a good idea to start with the menu system. The menu system is the most static of all devices. Its definition in general covers all operations the programmer wants the application to be able to do. It is quite easy to get a prototype running, consisting of only the menu system and some of the vital functions. Starting from this framework the application can be extended step by step.

In situations where an application is going to manipulate windows all in more or less the same way (like an editor defines text windows and a drawing program picture windows) it is a good idea to start first with one window only. Once the application works alright for one window, it is very easy to generalise the implementation to an arbitrary number of windows.

A representative application we have written is a window based editor, with all features that are commonly found in editors on Macintosh systems. The program state of the editor contains an UNQ data structure that administers the files. The majority of the program state is used for the proper administration of the text windows. For each window the content, position of the cursor, the current line that is being edited, the path name where the corresponding file can be found, selected text if there is one, tab width, the font used, some font metrics, the number of lines of the text, a flag stating whether the text has been changed since the last time it was saved and one stating if new lines should be automatically indented. The whole project consists of 18 implementation modules having a total number of lines of about 4600 (including comments). The editor works on both the Macintosh and the Sun4 system. Its runtime performance is very good and we use it in practice conveniently.

Among the other large applications we have written are a relational (DBase like) database and a (MacDraw like) drawing program. We are also working on an application with which devices can be created graphically. Such an application will mainly ease the definition of dialogues, which can become quite complex. The basic concept of the application is that once a device has been graphically created the algebraic type can be constructed that defines that device. Using nested I/O, complete interfaces can be easily tested by the graphical designer.

The libraries are mainly written in Concurrent Clean. The Macintosh implementation consists of 35 Clean modules having ± 7300 lines of Clean code and the Macintosh interface library another 19 modules having ± 2200 lines of assembly code. The implementation on the X Window System using the Open Look Interface Tool kit [8] consists of 35 Concurrent Clean modules of ± 5100 lines of code. The interface to the tool kit and the X Window System consists of 10 modules of ± 5100 lines of C code.

10. Discussion

Interactions can be arbitrarily complex due to the freedom of the type of the program state. However, because the program state holds all global data needed by the interaction, its content is accessible for all event handlers. This reintroduces (yet on a smaller scale) problems of reasoning about global variables.

In traditional functional languages access to data structures is either via pattern-matching or access functions on the data structure. Changing the type of a data structure implies change of the functions that pattern-match on that structure. The problem of changing data structures becomes apparent (and a real nuisance) when one needs to restructure the program state. In all cases access on data structures requires full knowledge of the structure. What is needed to increase program maintenance are data structures that can be accessed without exposing full knowledge of its content. Imperative languages have solved this problem by means of 'records' or 'structures'. We are currently investigating how to extend Concurrent Clean with record types.

The level of I/O that is offered to the application programmer is that of devices. Devices not only offer a high level of abstraction for the programmer, but are also very suited to serve as the basic building blocks for implementations on various platforms because programs are always unaware of the internal representation of devices. Each implementation on a specific platform can obey the look-and-feel that is demanded by that system. The concept of devices lends itself to the creation of new devices. Due to the object-oriented approach, adding and changing devices is not a hard job.

In our experience the restrictions imposed by the use of Unique Types did not frustrate the programming process. The demand that the program state must be UNQ generally only affects the outermost data constructor, while keeping the content shareable. Problems occur only when one wants to access UNQ components from an UNQ object. Ordinary access by introducing sharing is illegal so one either has to make an explicit copy or effectively remove the object to be accessed from its containing object.

11. 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. Programming I/O means specifying the required devices and event handlers. Most of the commonly used I/O facilities are supported. The system 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. We have found that it is surprisingly easy and good fun to write quite complex interactive applications.

Now that we have the ability to perform updates in a pure functional language all low-level I/O can be implemented. Together with the typical features of functional languages such as algebraic types and higher order functions we are now able to handle I/O better and more conveniently than with the special facilities generally offered by object oriented interface tool kits. The beast has turned into a prince.

Currently there are versions of the I/O library for the Open Look Interface Tool kit as well for OSF/Motif (both for Sun4 under X Windows) and there is a version for the Macintosh. These libraries provide the same interface, so a Clean program can run on either of these systems without any modification. Still, the resulting applications will obey the different look-and-feel for these machines. The library has been used to write several applications. Even large applications run at the required efficiency.

References

1. Achten PM. Operational Semantics of Clean Event I/O. Technical report - in preparation University of Nijmegen.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. Peyton Jones SL, Wadler Ph. Imperative Functional Programming. Extended Abstract, to appear in POPL 1993, University of Glasgow.
8. Pillich L. Portable Clean Event I/O. Department of informatics, Faculty of Mathematics and Informatics, University of Nijmegen. Master Thesis 230, July 1992.
9. Plasmeijer MJ, Eekelen van MCJD. Functional Programming and Parallel Graph Rewriting. Lecture notes. University of Nijmegen 1991/1992. To appear: Addison Wesley 1993.
10 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.
11. Smetsers JEW, Barendsen E, Eekelen van MCJD and Plasmeijer MJ. Guaranteeing destructive updatability through a type system with uniqueness information for graphs. *Submitted to the FPCA-1993*. University of Nijmegen.