The Game of Life: A CLEAN Programming Tutorial and Case Study

Anthony H. Dekker

Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511
e-mail: `tdekker@iscs.nus.sg`

May 31, 1994

Abstract

This report presents a tutorial for the CLEAN Functional Programming Language, in the form of a stepwise development of two programs for animating Conway's Game of Life. The power of the novel and elegant input/output and Graphical User Interface (GUI) facilities of CLEAN are demonstrated by this example, which contains most of the features required in useful applications software. Suggestions on suitable programming style are also included.

1 Introduction

We present a tutorial for the CLEAN Functional Programming Language produced at the University of Nijmegen [3, 5]. Two programs for animating Conway's Game of Life played on an infinite square grid are developed. These programs differ from the Life program supplied as part of the CLEAN system. They were written in CLEAN Version 0.8.4 as a learning exercise to display the various features of the language, and its usefulness for application programming. Programming suggestions in the CLEAN manual were used as a basis for the techniques used in this paper.

For a copy of the programs described here, contact the author at the address above. For information on CLEAN, contact the authors of the language at `clean@cs.kun.nl`. The language is available on Apple Macintosh, Sun3 and Sun4 (SPARC) systems. The author apologises to CLEAN's designers for any inadvertent injustices to the language which this report may contain.

This work was supported by National University of Singapore Research Project RP920614.

2 Why program in CLEAN?

Much has been written about the benefits of programming in a modern functional programming language. The use of a more powerful and concise notation allows shorter and more maintainable programs to be written than is the case for an imperative language. Compile-time type-checking in a functional language detects many more errors than for an imperative language, since omission of a subexpression is a type error, whereas omission of a command is not. Polymorphic typing allows generic functions to be written, and these can be re-used without alteration. Functional languages permit arguments to functions to be any kind of value, including other functions, thus allowing a wider range of generic functions. For example, a generic list sorting function would take as an argument the kind of comparison operator to be used, so that it could apply to every kind of list.

The CLEAN language is one of the first functional languages suitable for practical application programming. The compiler generates efficient code, so that the speed of CLEAN programs, though usually slower than C, is of the same order of magnitude. Real numbers are handled efficiently, and there is an elegant *unique type* mechanism for incorporating update-in-place file operations in a purely functional context. CLEAN has a module structure which allows large applications to be developed. Testing of functions is made easier by the provision of a kind of module which evaluates and prints expressions of arbitrary type. Finally, and perhaps most importantly, there is a large library of Graphical User Interface (GUI) facilities which allow useful applications to be developed for the Macintosh and X-windows environments. The same program will run in both environments (with slightly different look and feel for predefined GUI facilities). Windows, menus, dialog-boxes and their components are first-class objects of various predefined types, and so can be manipulated easily using functions. The equivalent of *methods* in object-oriented programming is obtained by including state-transition functions as components of these objects. The programming style required is novel, and thus requires some practice, but is much easier than X or Motif programming with an imperative language. Development of the CLEAN system is still in progress, so that further improvements in efficiency and usability can be expected.

3 Some Miscellaneous Functions

Our first module, entitled *lifemisc* and stored in the file `lifemisc.icl`, defines some basic list operations in CLEAN, for use in the other modules. Each CLEAN implementation module has a corresponding definition module which lists exported type and macro definitions and function type declarations. These definitions can be imported with the `IMPORT` declaration. Here we import CLEAN's standard library modules *deltaC, deltaI, deltaB, deltaM* and *deltaS*:

```
IMPLEMENTATION MODULE lifemisc;
IMPORT deltaC, deltaI, deltaB, deltaM, deltaS;
```

The constant `Huge` is macro-defined to be $2^{31}-1$ using the bit operations `SHIFTL%` and `NOT%` defined in the *deltaI* library. This constant can be used as a virtual infinity:

```
MACRO
   Huge -> NOT% (SHIFTL% 1 31);
```

The `RULE` keyword introduces a group of function type declarations and definitions. The `Min` function takes two integer arguments and produces an integer result. The `!` annotations indicate that `Min` is strict on both its arguments, i.e. the arguments are evaluated eagerly. Functions with strict `INT`, `BOOL`, `CHAR`, `REAL`, `STRING`, or `FILE` arguments are implemented more efficiently than lazy functions in CLEAN.

```
RULE
:: Min !INT !INT -> INT;
   Min x y -> x, IF < x y
           -> y;
```

The definition of `Min` uses a guarded right-hand side: `Min x y` rewrites to `x` if `x` is less than `y`, and to `y` otherwise (i.e. `Min x y` rewrites to the minimum of its two arguments). Notice that all binary operators in CLEAN are written in prefix notation, i.e. `< x y` rather than `x<y`. The `Max` function is defined in a similar way to `Min`:

```
:: Max !INT !INT -> INT;
   Max x y -> x, IF > x y
           -> y;
```

The well-known `Map` function is defined by pattern-matching on cases. CLEAN's list notation is similar to that of PROLOG. Notice that the arrow operator on types, like all binary operators, is written in prefix notation. Although a function must be declared with a fixed arity (in this case, 2) matching the arity in the definition, functions can be used as curried objects. For example, `Map` can be used with the type `=> (=> x y) (=> ![x] [y])`, which would be more readable written in infix notation as `(x => y) => ![x] => [y]`.

```
:: Map (=> x y) ![x] -> [y];
   Map f [] -> [];
   Map f [h|t] -> [f h | Map f t];
```

The list-length function `Len` can be efficiently implemented using the tail-recursive auxiliary function `LenAux` which has a strict accumulating parameter. This technique allows `Len` to execute in constant space.

```
:: Len ![x] -> INT;
   Len l -> LenAux 0 l;
```

The function `LenAux` is strict on its second (list) argument, since pattern-matching forces evaluation to weak head normal form. This strictness is inferred by CLEAN's strictness analyzer. The strictness of `Len` could be inferred from this, but `Len` must be declared strict for export purposes. The `++` operation increments its argument (and similarly `--` decrements it).

```
:: LenAux !INT [x] -> INT;
   LenAux i [] -> i;
   LenAux i [h|t] -> LenAux (++ i) t;
```

A similar technique can be used to concatenate a list of strings, generalising the binary string concatenation operation `+S`:

```
:: Concat ![STRING] -> STRING;
   Concat l -> ConcatAux "" l;

:: ConcatAux !STRING [STRING] -> STRING;
   ConcatAux s [] -> s;
   ConcatAux s [h|t] -> ConcatAux (+S s h) t;
```

The `Drop` function removes a specified number of elements from a list. Since rules are examined in sequence, `Drop 0 [h|t]` rewrites to `[h|t]` and not to `Drop (-- i) t`:

```
:: Drop !INT ![x] -> [x];
   Drop 0 l -> l;
   Drop i [] -> [];
   Drop i [h|t] -> Drop (-- i) t;
```

The `Take` operation is defined similarly. It is not strict on its list argument:

```
:: Take !INT [x] -> [x];
   Take 0 l -> [];
   Take i [] -> [];
   Take i [h|t] -> [h | Take (-- i) t];
```

The *lifemisc* definition module lists the exported macro and functions. It is stored in the file `lifemisc.dcl`.

```
DEFINITION MODULE lifemisc;
IMPORT deltaC, deltaI, deltaB, deltaM, deltaS;
```

By including the `IMPORT` declarations in the definition module, the library modules *deltaC, deltaI, deltaB, deltaM* and *deltaS* are automatically imported whenever *lifemisc* is imported.

```
MACRO
```

```
   Huge -> NOT% (SHIFTL% 1 31);
RULE
:: Min !INT !INT -> INT;
:: Max !INT !INT -> INT;
:: Map (=> x y) ![x] -> [y];
:: Len ![x] -> INT;
:: Concat ![STRING] -> STRING;
:: Drop !INT ![x] -> [x];
:: Take !INT [x] -> [x];
```

4 Representing Patterns for the Game of Life

The *lifepoint* module defines the representation of cells for the Game of Life. We have an infinite square grid of cells, which can each be *alive* or *dead*. A finite number of live cells are distributed fairly sparsely within a finite area of the grid, forming a Game-of-Life *pattern*. The area of the grid occupied by the pattern can grow without limit, although it always remains finite. A suitable representation for Game-of-Life patterns is a sorted list of live points (`PntList`), where each point (`Pnt`) is a pair of integers. The strictness annotations ensures that evaluation of the elements of the pair, and of the points in the list, is forced. This allows the CLEAN compiler to generate more efficient code.

```
IMPLEMENTATION MODULE lifepoint;
IMPORT lifemisc;

TYPE
:: Pnt -> (!INT,!INT);
:: PntList -> [!Pnt];
```

To represent the rectangle actually occupied by live cells, we use the `Rect` type, which stores the coordinates of the upper left and lower right corners of the rectangle. We assume coordinates increase going downwards and to the right, so that the rectangle occupied by live cells is specified by the pair of minimum coordinate values and the pair of maximum coordinate values. To assist in calculating these values, the `Lifestats` type is intended to record the minimum, total (for calculating average), and maximum values of x and y coordinates of live cells:

```
:: Rect -> (!Pnt, !Pnt);
:: Lifestats -> (!INT, !INT, !INT, !INT, !INT, !INT);
```

The rules for the Game of Life specify that live cells die if they have less than two or more than three live neighbours, and dead cells become live if they have precisely three live neighbours. To keep track of this information, the `Info` type stores a point, plus a boolean value to indicate if the point was live, and an integer to record the number of live neighbours:

```
:: Info -> (!INT,!INT,!BOOL,!INT);
:: InfoList -> [!Info];
```

The functions for manipulating points include a recursive function to test if two sorted lists of points are identical. The `&&` operator is an infix conditional-and for use in guards only. There is also a conditional-or (`||`) operator.

```
RULE
:: Same !PntList !PntList -> BOOL;
   Same [] [] -> TRUE;
   Same l  [] -> FALSE;
   Same [] l  -> FALSE;
   Same [(x,y)|t] [(x',y')|t'] -> Same t t', IF = x x' && = y y'
                               -> FALSE;
```

The `SwapPnt` function adds a point to a sorted list if it is absent, or deletes it if it is present. The notation `z:e` has the same meaning as `e`, but also defines the name `z` as a reference to `e` (implemented as a second pointer to `e`). Thus `l` denotes the entire input list `[(x',y')|t]`, and `u` denotes the first point in the list. The variable `p` denotes the result of adding the point `(x,y)` to front of `l` in the case that it is missing, which is the result of the function in two cases. The variable `q` denotes the result of adding the original head `u` to the front of a recursive function call. This result is also returned in two cases. The `==` symbol introduces a comment terminated by end-of-line:

```
:: SwapPnt !INT !INT !PntList -> PntList;
   SwapPnt x y [] -> [(x,y)];
   SwapPnt x y l:[u:(x',y')|t] -> p:[(x,y) | l], IF < y y'
                               -> q:[u | SwapPnt x y t], IF > y y'
                               -> p, IF < x x'  == now y=y'
                               -> q, IF > x x'
                               -> t;    == now y=y' & x=x', so delete u
```

The `SortPnt` function performs a (slow) insertion sort on a list of points, using `SwapPnt` for insertion. It is used only to create an initial sorted list:

```
:: SortPnt !PntList -> PntList;
   SortPnt [] -> [];
   SortPnt [(x,y)|t] -> SwapPnt x y (SortPnt t);
```

The `CropPnt` function crops the Game-of-Life grid to a finite rectangular viewing window $((a,b),(c,d))$, at the same time re-scaling coordinates so that (0,0) is the top left cell of the window. This function shows another form of local variable definition, where `p` is used first, and defined at the end of the rule:

```
:: CropPnt !INT !INT !INT !INT !PntList -> PntList;
   CropPnt a b c d [] -> [];
   CropPnt a b c d [(x,y)|t] -> p, IF < y b
                             -> [], IF > y d
                             -> p, IF < x a
                             -> p, IF > x c
                             -> [(- x a, - y b) | p],
                                p:CropPnt a b c d t;
```

The `StatPnt` function calculates the number of cells and other statistics for a list of points, using the tail-recursive auxiliary function `StatPntAux`. This auxiliary function takes as arguments the current minimum, total, and maximum values of x and y coordinates, and the number of points processed (initially 0). Since CLEAN has no unary negation operator, the initial value for the maximum x and y coordinates is defined using subtraction to be `(- 0 Huge)`:

```
:: StatPnt !PntList -> (!INT, !Lifestats);
   StatPnt l -> StatPntAux Huge 0 (- 0 Huge) Huge 0 (- 0 Huge) 0 l;

:: StatPntAux !INT !INT !INT !INT !INT !INT !INT PntList -> (!INT, !Lifestats);
   StatPntAux a b c d e f n [] -> (n, (a, b, c, d, e, f));
   StatPntAux a b c d e f n [(x,y)|t]
    -> StatPntAux (Min x a) (+ x b) (Max x c) (Min y d) (+ y e) (Max y f) (++ n) t;
```

The `StepInfo` function converts a list of points to an unsorted list of `Info` entries, one for each live cell and its eight neighbours. The entry for the live cell is given a true liveness flag and a live-neighbour count of zero, while the neighbours are given a false liveness flag and a live-neighbour count of one:

```
:: StepInfo PntList -> InfoList;
   StepInfo [] -> [];
   StepInfo [(x,y)|t] -> [(-- x, -- y, FALSE, 1),       (-- x, y,    FALSE, 1),
                          (-- x, ++ y, FALSE, 1),       (x,    -- y, FALSE, 1),
                             (x, y, TRUE , 0),          (x,    ++ y, FALSE, 1),
                          (++ x, -- y, FALSE, 1),       (++ x, y,    FALSE, 1),
                          (++ x, ++ y, FALSE, 1)   | StepInfo t];
```

The following three functions define a (fast) merge-sort `InfoSort` on a list of `Info` entries. The `InfoMerge` function merges two already sorted lists, combining `Info` entries that correspond to the same point, adding up the live-neighbour counts, and using logical `OR` to combine the liveness flags. The result of sorting is a list of entries corresponding to previously live cells or neighbours of previously live cells, with their live-neighbour counts. Such a use of merge-sort to combine together corresponding entries in a list is a useful functional programming technique.

```
:: InfoMerge InfoList InfoList -> InfoList;
   InfoMerge [] l -> l;
   InfoMerge l [] -> l;
   InfoMerge p:[u:(x,y,b,i)|t] q:[v:(x',y',b',j)|t']
        -> ls:[u | InfoMerge t q], IF < y y'
        -> gt:[v | InfoMerge p t'], IF > y y'
        -> ls, IF < x x'
        -> gt, IF > x x'
        -> [(x, y, OR b b', + i j) | InfoMerge t t'];

:: InfoSort InfoList -> InfoList;
   InfoSort l -> InfoSortAux (Len l) l;
```

The auxiliary function for the sorting process takes the length of the list as an argument, splits the list into two halves, sorts them recursively, and merges the results:

```
:: InfoSortAux !INT InfoList -> InfoList;
   InfoSortAux n l -> l, IF <= n 1
                   -> InfoMerge (InfoSortAux n2 (Take n2 l)) (InfoSortAux (- n n2) (Drop n2 l)),
                      n2: / n 2;
```

The `DoRule` function applies the rules of the Game of Life to the sorted `InfoList` to obtain the new list of points after one step of the game. The new live cells are those that were live with two or three live neighbours, or were dead with precisely three live neighbours:

```
:: DoRule InfoList -> PntList;
   DoRule [] -> [];
   DoRule [(x,y,b,i) | t]
        -> [(x,y) | DoRule t], IF = 3 i || (b && = 2 i)
        -> DoRule t;
```

Combining `StepInfo`, `InfoSort`, and `DoRule` gives the step function which takes a list of points and gives the new list after one step:

```
:: Step !PntList -> PntList;
   Step l -> DoRule (InfoSort (StepInfo l));
```

The *lifepoint* definition module (not shown) lists the exported types and functions, i.e. not including the auxiliary functions, or the `Info` types and operations, whose sole purpose was the definition of `Step`.

5 A Simple CLEAN Program

Using the above modules, we can write our first CLEAN program. The file `lifetest.icl` contains the program module *lifetest*:

```
MODULE lifetest;
IMPORT lifepoint;
```

A program module must contain a rule called `Start`, the execution of which defines the behaviour of the program. In this case the program calls `Step` with a list of four points. For modules of this kind, the CLEAN system is able to print out objects of any desired type, which allows for quick testing of modules by evaluating any desired expression.

```
RULE
:: Start -> PntList;
   Start -> Step [(1,1),(1,2),(1,3),(7,7)];
```

Execution of this program prints the correct new list of points, and the time taken:

```
[(0,2),(1,2),(2,2)]
Execution: 0.00  Garbage collection: 0.00  Total: 0.00
```

6 File Input and Output for the Game of Life

The *lifeio* module defines file input-output operations for the Game of Life. It exemplifies CLEAN's file I/O facilities defined in the library module *deltaFile*. A file object, of type `FILE`, is actually a file descriptor which contains a pointer into a physical disk file.

```
IMPLEMENTATION MODULE lifeio;
IMPORT deltaFile,lifepoint;

MACRO
   PSsquare -> 450;
```

The `FWriteL` function recursively writes a list of strings to a file, using the `FWriteS` primitive function to write a string. The function defined here is written in a functional style, with `f` referring to the initial file, and `f'` referring to the file after writing one string. However, the output to the file is actually performed as an update in place, in the usual way. This is possible using CLEAN's *unique type* mechanism.

An object of type `UNQ T` (such as `f`) cannot be shared, and an attempt to copy the object is detected as a type error. Since the object is used only once, it can be safely updated in place. The type of `FWriteS` is `!STRING => !UNQ FILE => UNQ FILE` (in infix notation), so that the result `f'` is also of unique type. The use of functions of type `UNQ FILE => UNQ FILE` essentially forces access to the file to be single-threaded. The strictness annotation on the file argument forces evaluation of file writing operations in the argument.

Unique types are closely related to linear types [2, 4]. The unique type mechanism offers an elegant handling of file update in the functional context, and could also be used to handle array operations with update in place.

```
RULE
:: FWriteL ![STRING] !UNQ FILE -> UNQ FILE;
   FWriteL [] f -> f;
   FWriteL [h|t] f -> FWriteL t f',
                      f': FWriteS h f;
```

Since the predefined file closing operation `FClose` in the current version of CLEAN will not operate on the standard files `StdIO` and `StdErr`, we provide a function with the same type to take its place. This function forces evaluation of all input and output operations on the file, but does not allow the file to be re-opened (and so we return a false success status). In CLEAN, opening and closing a file is treated conceptually as taking a file out of, or placing it back into, a file system of type `FILES`.

```
:: FCloseStd !UNQ FILE !FILES -> (!BOOL, !FILES);
   FCloseStd f g -> (FALSE, g);
```

The `SReadPtsList` function reads a list of points from a (possibly shared) text file, using non-unique shared-file operations. Such operations can only read, but not modify a file. The recursive loop terminates on end of file, or when an attempt to read an integer with `SFReadI` fails. Notice the sequence of file values: `f` is the initial file, `f'` is the file after having read one integer, and `f''` is the file after reading a second integer (i.e. `f''` contains a pointer into the file which points just after the second integer). It is this final file value which is passed to the recursive call.

A shared file object can be copied many times, and each copy can be used as the basis for read operations. Consequently, when a physical file is opened as a non-unique file (using `SFOpen`) it cannot be closed, since there may be many file objects pointing into the physical file.

```
:: SReadPtsList !FILE -> PntList;
   SReadPtsList f -> [], IF SFEnd f || NOT status1 || NOT status2
                  -> [(x,y) | SReadPtsList f''],
                     (status1, x, f'): SFReadI f,
                     (status2, y, f''): SFReadI f';
```

The `ReadPtsList` function is like `SReadPtsList`, but reads from an unshared text file, using unique-file operations. These require single-threaded use of the file, which complicates the function definition. However, single-threaded operations allow the final file object to be returned for later closing. Since unique files cannot be copied, they can be safely closed.

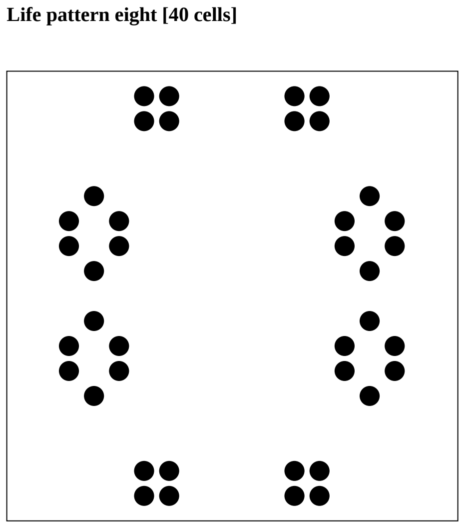

Figure 1: Output of `WritePS` function

To ensure single-threaded use, only shared-file operations can be applied to a file in guards. This is because it cannot be decided if a guard will be executed, and also because a guard returns only a boolean result, and cannot return a file object. Thus the test for end of file must be done using the shared-file operation `SFEnd`.

```
:: ReadPtsList !UNQ FILE -> (!PntList, !UNQ FILE);
   ReadPtsList f -> ([], f), IF SFEnd f
                 -> ([], f'), IF NOT status1
                 -> ([], f''), IF NOT status2
                 -> ([(x,y)|rest], f'''),
                    (status1, x, f'): FReadI f,
                    (status2, y, f''): FReadI f',
                    (rest,f'''): ReadPtsList f'';
```

The `WritePtsList` function writes a list of points to a unique text file. The sequence of operations is to write `x` to `f`, then write a space, giving `f'`, and then write `y` and a newline, giving `f''` which is passed to the recursive call.

```
:: WritePtsList !PntList !UNQ FILE -> UNQ FILE;
   WritePtsList [] f -> f;
   WritePtsList [(x,y)|t] f -> WritePtsList t f'',
                               f': FWriteC ' ' (FWriteI x f),
                               f'': FWriteC '\n' (FWriteI y f');
```

The `WritePS` function is more sophisticated, writing an Encapsulated PostScript [1] picture of a list of points occupying a given rectangle, with a string to print as heading. It uses three auxiliary functions, after calculating the radius of the circles used to indicate live cells. An example Encapsulated PostScript file generated by this function was imported directly into this document as Figure 1. The `PSsquare` macro defined at the start of the module indicates the size (in PostScript units) of the square in which the pattern is drawn.

```
:: WritePS !STRING !Rect !PntList !UNQ FILE -> UNQ FILE;
   WritePS s ((a,b),(c,d)) l f
        -> f''',
           f': PSheader s radius f,
           f'': PSPtsList xoffset yoffset a b size l f',
           f''': PStrailer f'',
           width: + 2 (- c a),
           height: + 2 (- d b),
           size: Max 2 (Min (/ PSsquare height) (/ PSsquare width)),
           radius: Max 1 (/ (* 8 size) 20),
           actualwidth: * size (- c a),
           actualheight: * size (- d b),
           xoffset: / (- PSsquare actualwidth) 2,
           zoffset: / (- PSsquare actualheight) 2,
           yoffset: - PSsquare zoffset;
```

The first auxiliary function uses `FWriteL` to write a list of strings to form an Encapsulated PostScript header. This provides a convenient way of performing formatted output. The `ITOS` primitive function converts an integer to a string.

```
:: PSheader !STRING !INT !UNQ FILE -> UNQ FILE;
   PSheader s radius f
    -> FWriteL ["%!PS-Adobe-2.0 EPSF-2.0\n",
                "%%Title: ", s, "\n",
                "%%Creator: LIFE package by A. H. Dekker\n",
                "%%BoundingBox: 19 99 ", ITOS (+ 21 PSsquare), " ", ITOS (+ 171 PSsquare), "\n",
```

```
                    "%%DocumentFonts: Times-Bold\n",
                    "%%EndProlog\n\n",
                    "/origstate save def\n",        "20 dict begin\n",
                    "20 100 translate\n\n",         "0 ", ITOS (+ 50 PSsquare), " moveto\n",
                    "/Times-Bold findfont 20 scalefont setfont\n",
                    "(", s, ") show\n",             "1 setlinewidth 0 0 moveto\n",
                    ITOS PSsquare, " 0 rlineto 0 ", ITOS PSsquare, " rlineto ",
                    ITOS (- 0 PSsquare), " 0 rlineto closepath stroke\n",
                    "/dot {", ITOS radius, " 0 360 arc fill} bind def\n\n"] f;
```

The second auxiliary function recursively writes a list of points in a similar way to `WritePtsList`:

```
:: PSPtsList !INT !INT !INT !INT !INT PntList !UNQ FILE -> UNQ FILE;
   PSPtsList xoff yoff a b size [] f -> f;
   PSPtsList xoff yoff a b size [(x,y)|t] f
        -> PSPtsList xoff yoff a b size t f'',
           f': FWriteC ' ' (FWriteI (+ xoff (* size (- x a))) f),
           f'': FWriteS " dot\n" (FWriteI (- yoff (* size (- y b))) f');
```

The final auxiliary function writes an Encapsulated PostScript trailer:

```
:: PStrailer !UNQ FILE -> UNQ FILE;
   PStrailer f
    -> FWriteL ["\n", "showpage\n", "end\n", "origstate restore\n"] f;
```

The *lifeio* definition module (not shown) lists the five exported functions of the module.

7 A File Conversion Program

Our second CLEAN program, *lifeconvert*, indicates the format for character-based application programs. The `Start` rule is not an expression to be printed, but a state transition function to be applied to the outside world. This program converts a file containing a list of points to the PostScript format described in section 6.

The `WORLD` abstract data type describes the world outside the program, containing among other things the file system. The Unix command-line argument vector would logically also be part of this abstract data type, but cannot be accessed by CLEAN at present.

```
MODULE lifeconvert;
IMPORT lifeio;

RULE
:: Start UNQ WORLD -> UNQ WORLD;
```

The result of this program is an `ABORT` message if there are errors in opening or closing files. The first step in calculating the result `world''` of the program is to use `OpenFiles` to extract the file system from the world, and in turn to extract the standard input/output file `StdIO` from the file system. The variable `f1` contains the file descriptor for this file.

```
   Start world
     -> ABORT (Concat ["Can't open '", input, "'\n"]), IF NOT okinput
     -> ABORT (Concat ["No cells found in '", input, "'\n"]), IF = numcells 0
     -> ABORT (Concat ["Can't open '", output, "'\n"]), IF NOT okopen
     -> ABORT (Concat ["Can't close '", output, "'\n"]), IF NOT okclose
     -> world'',
        (filesystem1, world'): OpenFiles world,
        (f1, filesystem2): StdIO filesystem1,
```

The standard input/output file is then modified using single-threaded operations which output a prompt, and read a line of input. This line of input is a string which contains a terminating new-line character, so the `DropNewLine` macro defined at the end of the program is used to remove the last character of the string. The style of functional programming needed here seems a little strange at first, but provides a useful discipline for the programmer. We use `f1`, `f2`, etc. for the thread of file values for the standard input/output file, just as we use `filesystem1`, `filesystem2`, etc. for the thread of file system values.

```
        f2: FWriteS "Input file name: " f1,
        (inputline, f3): FReadLine f2,
        input: DropNewLine inputline,
```

The string read from the input is used as a file name for opening a non-unique input file `g`. The constant `FReadText` opens the file as a text file for reading only. The list of points in the input file is read using the shared-file operation defined in section 6, and the number of cells and minimum and maximum coordinates are calculated. Notice that tuples cannot be nested in the left-hand side of local definitions, i.e. a definition of the form `(x,(y,z)):e` is not legal.

```
        (okinput, g, filesystem3): SFOpen input FReadText filesystem2,
        points: SReadPtsList g,
        (numcells, statistics): StatPnt points,
        (a, totx, c, b, toty, d): statistics,
```

The output file name is then read and used to open the output file `h` for writing:

```
        f4: FWriteS "Output file name: " f3,
        (outputline, f5): FReadLine f4,
```

```
            output: DropNewLine outputline,
            (okopen, h, filesystem4): FOpen output FWriteText filesystem3,
```

We then write the PostScript file, using an appropriate header message, and close the output file. The `WritePS` function requires the rectangle $((a,b),(c,d))$ occupied by the points.

```
            cellcount: Concat [" [", ITOS numcells, " cells]"],
            message: Concat ["Life pattern ", input, cellcount],
            h': WritePS message ((a,b),(c,d)) points h,
            (okclose, filesystem5): FClose h' filesystem4,
```

Finally, we write a message to the standard output, and use our `FCloseStd` function to force the write to be executed (the dummy status is ignored). Placing the final file system back into the world gives the final result of the program. In spite of its name, `CloseFiles` does not close any open files, it merely places the file system back into the world.

```
            f6: FWriteS (Concat [input, " -> ", output, cellcount, "\n"]) f5,
            (dummy, filesystem6): FCloseStd f6 filesystem5,
            world'': CloseFiles filesystem6 world';
```

Removing the last character of a string is done with the following macro, which selects characters $0 \ldots (n-2)$ of a string of length n:

```
MACRO
   DropNewLine s -> SLICE s 0 (- (LENGTH s) 2);
```

Execution of this program is as follows, with the output shown in Figure 1. At present CLEAN prints a redundant `65536` which corresponds to the final world object.

```
Input file name: eight
Output file name: PATTERN
eight -> PATTERN [40 cells]
65536
Execution: 0.02  Garbage collection: 0.00  Total: 0.02
```

The present version of CLEAN cannot directly print files, but on a Macintosh we can combine the PostScript output with existing 'drag-and-drop' printing tools. On a Unix system we can write a file (with a name constructed from the input file name and the current time) to a spooling directory, and use a shell script to print it. This provides the same functionality as being able to directly print files.

8 A GUI-Based Conversion Program

Our third CLEAN program, *winconvert*, indicates the format for GUI-based application programs. Such programs are again state transition functions on the world, but are based around two special types. These are a user-defined internal state (in this case `State` is the file system) and a system-defined state which is parameterised on the user-defined state and must be unique. The type declarations below establish `State` as shorthand for `UNQ FILES`, and `IO` as shorthand for `UNQ IOState State`.

On the Macintosh, this program must be compiled with the 'No Console' option. This generates a runnable Macintosh application. It should be noted that CLEAN programs on the Macintosh can only manipulate text and data files, and cannot access file icons or other items in a file's resource fork.

```
MODULE winconvert;
IMPORT lifeio;
IMPORT deltaEventIO, deltaIOSystem, deltaFileSelect;
FROM deltaDialog IMPORT Beep, OpenNotice;
TYPE
:: UNQ State -> FILES;
:: UNQ IO -> IOState State;
```

All GUI-based CLEAN programs require a special *events* object to be extracted from the world, and in this case we also need the file system, since we are using it as an internal user-defined state. The `StartIO` operator takes the menu system, the initial user-defined state and the events object, and starts the interactive system. On termination of interaction, it returns the final value of the user-defined state and the final events object, both of which we put back into the world.

```
RULE
:: Start UNQ WORLD -> UNQ WORLD;
   Start world
     -> world''',
        (filesystem, world'): OpenFiles world,
        (events, world''): OpenEvents world',
        (filesystem', events'): StartIO [menus] filesystem [] events,
        world''': CloseFiles filesystem' (CloseEvents events' world''),
```

The menu system consists of only one menu, entitled **File**, which is a pull-down menu containing two items. Notice that the entire specification is a term of an algebraic data type, containing titles, menu command-key short-cuts, ability to be selected, and *method* functions which are executed when a menu item is selected. These methods are state transition functions on the two internal states.

```
            menus: MenuSystem [filemenu],
            filemenu: PullDownMenu 1 "File" Able [
                  MenuItem 1 "Convert" (Key 'C') Able ConvertFunction,
```

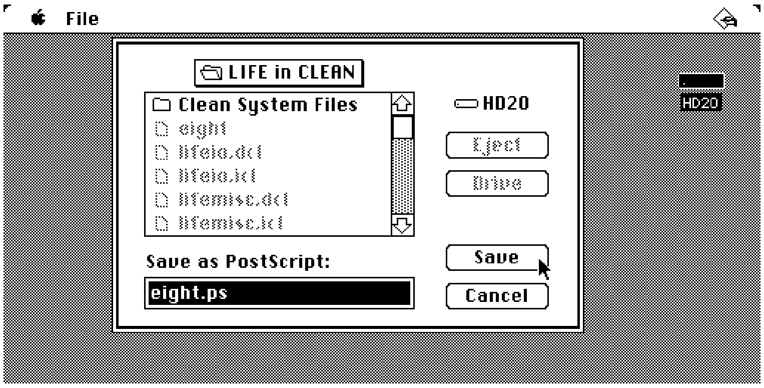

Figure 2: Predefined output-file selection dialog box on Macintosh

```
                    MenuItem 2 "Quit" (Key 'Q') Able QuitFunction];
```

The method for the **Quit** menu item alters the system-defined state using `QuitIO`, which causes the execution of `StartIO` to terminate, and hence stops the program:

```
:: QuitFunction State IO -> (State, IO);
   QuitFunction s io -> (s, QuitIO io);
```

The method for the **Convert** menu item produces an error message in a dialog box if there are problems in opening or closing files. These dialog boxes are opened by applying a function to the system-defined state. Care is required in returning the correct file system `s2`, `s3`, etc. for each case, but the type-checker will detect failure to do this correctly.

```
:: ConvertFunction State IO -> (State, IO);
   ConvertFunction s1 io
     -> (s2, io'), IF NOT openselected
     -> (s3, Error (+S "Can't open file " input) io'), IF NOT okinput
     -> (s4, Error (+S "No cells found in " input) io'), IF = numcells 0
     -> (s5, io''), IF NOT saveselected
     -> (s6, Error (+S "Can't open file " output) io''), IF NOT okopen
     -> (s7, Error (+S "Can't close file " output) io''), IF NOT okclose
     -> (s7, Message ["File Converted", cellcount] io''),
```

The predefined input-file selection dialog box is used to select an input file name. This box allows the user to browse through the file system, which is thus provided as an argument. The result is a boolean value (`openselected`) which indicates if the user pressed the **Open** or **Cancel** buttons, the file name selected, and new values of the file system and system-defined state. If the **Cancel** button was pressed, the guarded rule above terminates `ConvertFunction` without producing a message.

```
          (openselected, input, s2, io'): SelectInputFile s1 io,
```

The selected file is then opened, read, and closed. This is just as in the previous program in section 7, but the file is opened as a unique file and read using `ReadPtsList`, so that it can be closed. Since only a limited number of files can be open at one time, it is important that a program which repeatedly opens files also closes them. Notice that for this program we ignore errors in closing the input file (i.e. we do not use the boolean result `dummy`).

```
          (okinput, g, s3): FOpen input FReadText s2,
          (points, g'): ReadPtsList g,
          (numcells, statistics): StatPnt points,
          (a, totx, c, b, toty, d): statistics,
          (dummy, s4): FClose g' s3,
```

The output file name is selected using the predefined output-file selection dialog box. This box again allows the user to browse through the file system, but also takes two string arguments: the 'Save as PostScript:' prompt, and the default output file name. The result is similar to that of `SelectInputFile`. The format of this dialog box on the Macintosh is shown in Figure 2. This predefined dialog box automatically requests confirmation if the selected output file already exists.

```
          (saveselected, output, s5, io''): SelectOutputFile "Save as PostScript:" (+S input ".ps") s4 io',
```

The output file is opened, written, and closed as in section 7:

```
          (okopen, h, s6): FOpen output FWriteText s5,
          cellcount: Concat ["[", ITOS numcells, " cells]"],
          message: Concat ["Life pattern ", input, " ", cellcount],
          h': WritePS message ((a,b),(c,d)) points h,
          (okclose, s7): FClose h' s6;
```

Messages are produced using *notice* boxes, which are simple dialog boxes of predefined format, containing a number of lines of text, specified as a list of strings, and one or more *buttons*, each with an identifying number and label. Figure 3 shows the confirmation notice produced by `Message ["File Converted", cellcount]` after successfully completing input and output. The `OpenNotice` function returns the identifying number of the button pressed by the user, which is of no interest when there is only one button. Notices, like the file-selector dialog boxes, are *modal* – they must be dealt with by the user before any other action can take place.

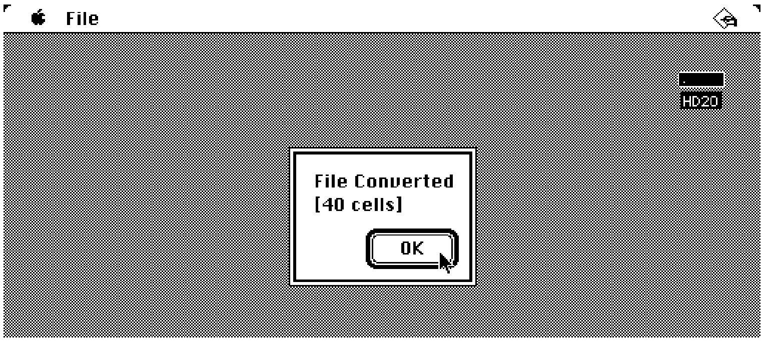

Figure 3: Confirmation notice box on Macintosh

```
:: Message [STRING] IO -> IO;
   Message mess io -> io',
                      (button, io'): OpenNotice notice io,
                      notice: Notice mess (NoticeButton 1 "OK") [];
```

Error messages are also produced using the `Message` function, after applying the self-explanatory `Beep` function to the system-defined state:

```
:: Error STRING IO -> IO;
   Error str io -> Message ["AN ERROR HAS OCCURRED", "", str] (Beep io);
```

9 An Abstract Data Type for Animating the Game of Life

To animate the Game of Life, we introduce the `Lifestate` abstract data type, defined in the *lifestate* module. We will use this abstract data type for both character-based and GUI-based animations.

```
IMPLEMENTATION MODULE lifestate;
IMPORT lifepoint;
```

This abstract data type is implemented as a tuple which contains all the information required for animation. This includes the number of steps performed, the number of live cells, and the status of the pattern. This status is an element of an enumerated type, which is `DEAD` for patterns containing no live cells, `STABLE` for patterns which do not change when a step is performed, and `DYNAMIC` otherwise:

```
TYPE
:: STEPNUM -> INT;
:: Lifestatus -> DYNAMIC | STABLE | DEAD;
```

The tuple also contains the coordinates of the upper left corner of the viewing window, the width and height of the viewing window, the statistics for the pattern as discussed in section 4, the sorted list of points constituting the pattern, and the sorted list of points visible in the viewing window. Laziness is essential in this last element, since we do not want to compute it if it is not needed.

```
:: WIDTH -> INT;
:: HEIGHT -> INT;
:: Lifestate -> (!STEPNUM, !INT, !Lifestatus, !Pnt, !WIDTH, !HEIGHT, !Lifestats, PntList, PntList);
```

The `Emptystate` function initialises an empty pattern:

```
RULE
:: Emptystate !WIDTH !HEIGHT -> Lifestate;
   Emptystate w h -> (0,0,DEAD,(0,0),w,h,(0,0,0,0,0,0),[],[]);
```

There are a number of projection functions for extracting the components of the tuple. Since `Lifestate` will be declared as an abstract data type, it will only be accessible via the operations in this module. Since a `DEAD` pattern is a special case of a `STABLE` one, the `Stable` function returns true for `DEAD` patterns.

```
:: Stepno !Lifestate -> STEPNUM;
   Stepno (step, n, status, upleft, w, h, stats, pts, crp) -> step;

:: Numcells !Lifestate -> INT;
   Numcells (step, n, status, upleft, w, h, stats, pts, crp) -> n;

:: Status !Lifestate -> Lifestatus;
   Status (step, n, status, upleft, w, h, stats, pts, crp) -> status;

:: Dead !Lifestate -> BOOL;
   Dead (step, n, DEAD, upleft, w, h, stats, pts, crp) -> TRUE;
   Dead (step, n, status, upleft, w, h, stats, pts, crp) -> FALSE;

:: Stable !Lifestate -> BOOL;
   Stable (step, n, STABLE, upleft, w, h, stats, pts, crp) -> TRUE;
```

```
   Stable (step, n, DEAD, upleft, w, h, stats, pts, crp) -> TRUE;
   Stable (step, n, status, upleft, w, h, stats, pts, crp) -> FALSE;

:: Width !Lifestate -> WIDTH;
   Width (step, n, status, upleft, w, h, stats, pts, crp) -> w;

:: Height !Lifestate -> HEIGHT;
   Height (step, n, status, upleft, w, h, stats, pts, crp) -> h;

:: Allpoints !Lifestate -> PntList;
   Allpoints (step, n, status, upleft, w, h, stats, pts, crp) -> pts;

:: Croppedpoints !Lifestate -> PntList;
   Croppedpoints (step, n, status, upleft, w, h, stats, pts, crp) -> crp;
```

The `Picrect` function calculates the rectangle $((a, b), (c, d))$ occupied by the points in the pattern, for use with e.g. `WritePS`:

```
:: Picrect !Lifestate -> Rect;
   Picrect (step, n, status, upleft, w, h, (minx,totx,maxx,miny,toty,maxy), pts, crp)
     -> ((minx,miny), (maxx,maxy));
```

On the other hand, the `Winrect` function calculates the rectangle $((a, b), (c, d))$ occupied by the viewing window:

```
:: Winrect !Lifestate -> Rect;
   Winrect (step, n, status, upleft:(a,b), w, h, stats, pts, crp)
     -> (upleft, (c,d)),
        c: -- (+ a w),
        d: -- (+ b h);
```

The `CentOfGrav` function calculates the average value of the coordinates of live cells, giving the 'centre of gravity' of the picture:

```
:: CentOfGrav !Lifestate -> Pnt;
   CentOfGrav (step, n, status, upleft, w, h, (minx,totx,maxx,miny,toty,maxy), pts, crp)
     -> (0,0), IF = n 0
     -> (/ totx n, / toty n);
```

The coordinates of the centre of the viewing window are calculated by `ActualCent`:

```
:: ActualCent !Lifestate -> Pnt;
   ActualCent (step, n, status, (a,b), w, h, stats, pts, crp)
     -> (+ a (/ w 2), + b (/ h 2));
```

The `CentreAt` function moves the viewing window so that it is centred on a given point (x, y). This requires re-cropping the pattern to fit the new window:

```
:: CentreAt !Pnt !Lifestate -> Lifestate;
   CentreAt (x,y) (step, n, status, upleft, w, h, stats, pts, crp)
     -> (step,  n, status, (a,b), w, h, stats, pts, newcrp),
        a: - x (/ w 2),
        b: - y (/ h 2),
        newcrp: CropPnt a b c d pts,
        c: -- (+ a w),
        d: -- (+ b h);
```

We can use this operation to define a function which centres the viewing window on the 'center of gravity' of the pattern:

```
:: GoCentre !Lifestate -> Lifestate;
   GoCentre s -> s, IF = 0 (Numcells s)
              -> CentreAt (CentOfGrav s) s;
```

The `Resize` function alters the width and height of the viewing window, while maintaining the coordinates of the centre. The `CentreAt` operation will re-crop the pattern to fit the new window:

```
:: Resize !WIDTH !HEIGHT !Lifestate -> Lifestate;
   Resize w h s:(step, n, status, upleft, oldw, oldh, stats, pts, crp)
     -> CentreAt centre (step, n, status, upleft, w, h, stats, pts, crp),
        centre: ActualCent s;
```

The `Loadpoint` function places a list of points representing a new pattern into the tuple. This list must be sorted, and the statistics recalculated. The step count must also be reset to zero. The `GoCentre` function will centre the viewing window on the new pattern and re-crop:

```
:: Loadpoint !PntList !Lifestate -> Lifestate;
   Loadpoint l (step, n, status, upleft, w, h, stats, pts, crp)
     -> Emptystate w h, IF = m 0
     -> GoCentre (0, m, DYNAMIC, upleft,  w, h, newstats, newpts, []),
        newpts: SortPnt l,
        (m, newstats): StatPnt newpts;
```

The `Changepoint` function inverts the state of a cell in the pattern, specified using absolute coordinates, or relative to the upper left corner of the viewing window. An enumerated type `PntRefKind` is used to identify the two types of call:

```
TYPE
:: PntRefKind -> RELATIVE | ABSOLUTE;

RULE
:: Changepoint !PntRefKind !Pnt !Lifestate -> Lifestate;
   Changepoint ABSOLUTE (x,y) s:(step, n, status, upleft:(a,b), w, h, stats, pts, crp)
     -> ChangepointAux x y (- x a) (- y b) s;
   Changepoint RELATIVE (x,y) s:(step, n, status, upleft:(a,b), w, h, stats, pts, crp)
     -> ChangepointAux (+ a x) (+ b y) x y s;
```

The auxiliary function inverts the cell in the pattern, and also in the viewing window if it falls inside the window area. The statistics are recalculated, and the step count reset to zero:

```
:: ChangepointAux !INT !INT !INT !INT !Lifestate -> Lifestate;
   ChangepointAux absx absy relx rely
        (step, n, status, upleft, w, h, stats, pts, crp)
             -> Emptystate w h, IF = m 0
             -> (0, m, DYNAMIC, upleft, w, h, newstats, newpts, newcrp),
                       IF >= relx 0 && >= rely 0 && < relx w && < rely h
             -> (0, m, DYNAMIC, upleft, w, h, newstats, newpts, crp),
                newpts: SwapPnt absx absy pts,
                newcrp: SwapPnt relx rely crp,
                (m, newstats): StatPnt newpts;
```

The most important function is `Lifestep` which increments the step count and calculates a new list of points using the `Step` function from section 4. The statistics are recalculated, and the points are re-cropped to fit the viewing window. However, if the step resulted in no change to the list of points, these alterations are not performed, and only the status is changed to indicate that the pattern has become stable:

```
:: Lifestep !Lifestate -> Lifestate;
   Lifestep (step, n, status, upleft:(a,b), w, h, stats, pts, crp)
     -> (++ step, 0, DEAD, upleft, w, h, newstats, [], []), IF = m 0
     -> (step, n, STABLE, upleft, w, h, stats, pts, crp), IF Same pts newpts
     -> (++ step, m, DYNAMIC, upleft, w, h, newstats, newpts, newcrp),
        newpts: Step pts,
        newcrp: CropPnt a b c d newpts,
        c: -- (+ a w),
        d: -- (+ b h),
        (m, newstats): StatPnt newpts;
```

A variation of `Lifestep` is `Trackstep`, which follows the step by readjusting the viewing window, if the boolean argument is true. Since `GoCentre` re-crops to fit the viewing window, the cropped list produced by `Lifestep` is lazily replaced without being evaluated:

```
TYPE
:: TRACKMODE -> BOOL;

RULE
:: Trackstep !TRACKMODE !Lifestate -> Lifestate;
   Trackstep track s -> GoCentre s', IF track
                     -> s',
                        s': Lifestep s;
```

The *lifestate* definition module (not shown) exports the name of the `Lifestate` type as an abstract data type, using `ABSTYPE ::Lifestate`. Its implementation as a tuple is hidden, so that the type can only be accessed using the exported functions.

10 An Animation Program

We are now in a position to write a LIFE animation program, *lifeanimate*, using character-based rather than GUI facilities. The structure of the program is similar to that of the file conversion program in section 7. It will use the abstract data type `Lifestate` defined in section 9.

```
MODULE lifeanimate;
IMPORT lifestate,lifeio;
```

The algebraic data type `Command` defines the possible interactive commands: move the viewing window to the centre of gravity of the picture (`GoToCentre`), erase the picture (`Erase`), print a help message (`Help`), quit the program (`Quit`), run the animation for a number of steps (`Run`), centre the viewing window on a specified point (`GoTo`), invert the state of a specified cell (`ChgPnt`), load a pattern from a file (`Load`), store a pattern to a file (`Store`) or write an Encapsulated PostScript picture of a pattern (`PostScript`). The dummy command `Illegal` is used to represent illegal user responses, and contains a string which represents an error message.

```
TYPE
:: FILENAME -> STRING;
:: Command -> GoToCentre | Erase | Help | Quit | Illegal !STRING | Run !TRACKMODE !INT |
              GoTo !Pnt | ChgPnt !Pnt | Load !FILENAME | Store !FILENAME | Pscript !FILENAME;
```

The following macros define the size of the viewing window (38×17), and the strings used to print live and dead cells on a character-based terminal.

```
MACRO
   W -> 38;
   H -> 17;
   Star -> "* ";
   Blank -> "  ";
```

CLEAN allows the user to provide rewrite rules on the constructors of an algebraic data type. We use this facility to rewrite terms of the `Command` type to `Illegal` in the case of errors. This is a useful general technique: it simplifies the interactive input/output by distributing some of the error-checking.

```
RULE
:: Run !TRACKMODE !INT -> Command;
   Run track i -> Illegal "Must have i>0", IF <= i 0;

:: Load !FILENAME -> Command;
   Load "" -> Illegal "Empty file name";

:: Store !FILENAME -> Command;
   Store "" -> Illegal "Empty file name";

:: Pscript !FILENAME -> Command;
   Pscript "" -> Illegal "Empty file name";
```

Two useful functions for input handling are `SkipLine`, which skips to the end of the input line, and `SkipToSpace`, which skips to the next white space, returning a boolean flag if end-of-line was reached:

```
:: SkipLine !UNQ FILE -> UNQ FILE;
   SkipLine f -> f',
                 (s, f'): FReadLine f;

:: SkipToSpace !UNQ FILE -> (!BOOL, !UNQ FILE);
   SkipToSpace f -> (TRUE, f'), IF =C ch '\n'
                 -> (FALSE, f'), IF =C ch ' ' || =C ch '\t'
                 -> SkipToSpace f',
                    (ok, ch, f'): FReadC f;
```

The next two functions print out in short and long form the possible user commands:

```
:: ShortCommands !UNQ FILE -> UNQ FILE;
   ShortCommands f
     -> FWriteS s f,
        s: "H(elp  Q(uit  R i  T i  G x y  C  E  + x y  L f  S f  P f\n";

:: ShowCommands !UNQ FILE -> UNQ FILE;
   ShowCommands f
    -> FWriteL msg f,
       msg: ["Legal Commands Are:\n",
             "\tH(elp       - display this Help message\n",
             "\tQ(uit       - quit the program\n",
             "\tR(un  i     - run for i steps\n",
             "\tT(rack  i   - run for i steps in tracking mode\n",
             "\tG(oto  x  y  - move window so (x,y) is centred\n",
             "\tC(entre     - move window to centre of picture\n",
             "\tE(rase      - erase all cells\n",
             "\t+  x  y     - toggle state of (x,y) cell\n",
             "\tL(oad  f    - load picture from file f\n",
             "\tS(tore  f   - store picture to file f\n",
             "\tP(ostscr  f  - store PostScript picture to file f\n",
             "\n"];
```

The `Readcommand` function reads a command from an input file. Care must be taken to ensure single-threading. For example, access in a guard to a variable such as `ok` which is the result of a read operation forces that read operation to occur, in which case the value of the file before the read operation becomes inaccessible. The unique type mechanism ensures single-threading, but it is inadvisable to rely on this mechanism too heavily. Distributing some of the error-checking, as discussed above, assists in producing single-threaded code.

```
:: Readcommand !UNQ FILE -> (!Command, !UNQ FILE);
   Readcommand f
     -> (Illegal "Blank Line", f'),    IF =C ch '\n'
     -> (GoToCentre, SkipLine f'),     IF =C ch 'C' || =C ch 'c'
     -> (Erase, SkipLine f'),          IF =C ch 'E' || =C ch 'e'
     -> (Help, SkipLine f'),           IF =C ch 'H' || =C ch 'h'
     -> (Quit, SkipLine f'),           IF =C ch 'Q' || =C ch 'q'
```

Up to this point, the first input character `ch` has been read, which makes `f'` the current file value. Examining the variable `eol` forces a `SkipToSpace` operation, making `f''` the current file variable. From this, the file variable `h` is obtained by reading a filename `s'`:

```
-> (Illegal "No Arguments", f''), IF eol
-> (Load s', h),                  IF =C ch 'L' || =C ch 'l'
-> (Store s', h),                 IF =C ch 'S' || =C ch 's'
-> (Pscript s', h),               IF =C ch 'P' || =C ch 'p'
```

Examining `ok` in a guard forces a number to be read. To ensure single-threading it is necessary to order the rules so that all subsequent rules permit a number to be read:

```
-> (Run TRUE x, SkipLine g),      IF (=C ch 'T' || =C ch 't') && ok
-> (Run FALSE x, SkipLine g),     IF (=C ch 'R' || =C ch 'r') && ok
-> (GoTo (x,y), SkipLine g'),     IF (=C ch 'G' || =C ch 'g') && ok && ok'
-> (ChgPnt (x,y), SkipLine g'),   IF =C ch '+' && ok && ok'
-> (Illegal "Illegal Input", SkipLine g'),
```

For the default rule, the current file value must be taken as `g'` since one or two numbers may already have been read as a result of examining `ok` or `ok'` in failed guards. The local definitions for `Readcommand` are:

```
(dummy, ch, f'): FReadC (FWriteS "> " f),
(eol, f''): SkipToSpace f',
(s, h): FReadLine f'',
s': GetFileName 0 s,
(ok, x, g): FReadI f'',
(ok', y, g'): FReadI g;
```

The `GetFileName` function strips away white space from around a filename in an input string. The function `GetFileNameAux` is called when the beginning of the filename is found: it searches for the end of the filename and extracts it from the string:

```
:: GetFileName !INT !STRING -> STRING;
   GetFileName i s
     -> "", IF =C ch '\n'
     -> GetFileName (++ i) s, IF =C ch ' ' || =C ch '\t'
     -> GetFileNameAux i (++ i) s,
        ch: INDEX s i;

:: GetFileNameAux !INT !INT !STRING -> STRING;
   GetFileNameAux start j s
     -> SLICE s start (-- j), IF =C ch ' ' || =C ch '\t' || =C ch '\n'
     -> GetFileNameAux start (++ j) s,
        ch: INDEX s j;
```

Corresponding to the `Help` command, the following function prints a help message, and then calls `ShowCommands`:

```
:: DoHelp !UNQ FILE -> UNQ FILE;
   DoHelp f
     -> ShowCommands f',
        f': FWriteL msg f,
        msg: ["Display shows step no, number of cells, picture size\n",
              "and centre, and window size.\n",
              "\n",
              "Initial state is DEAD (no cells) - to create a pattern\n",
              "use '+ x y' repeatedly, or 'L filename'\n",
              "\n"];
```

Corresponding to illegal commands, the following function prints an error message, and then calls `ShowCommands`:

```
:: DoIllegal !STRING !UNQ FILE -> UNQ FILE;
   DoIllegal mess f
     -> ShowCommands f',
        f': FWriteL msg f,
        msg: ["YOU HAVE ENTERED AN INVALID COMMAND (", mess, ")\n\n"];
```

The `Statusline` function gives a list of strings to be printed as part of the description of the current Life pattern. It shows the coordinates and properties of the pattern and the viewing window. The ability to manipulate strings in this way simplifies input/output.

```
:: Statusline Lifestate -> [STRING];
   Statusline s
     -> ["#", ITOS (Stepno s),"    DEAD\n\n"], IF Dead s
     -> ["#", ITOS (Stepno s),"  ", ITOS (Numcells s), " Cells  ", Shstatus (Status s),
         "  Pic (", ITOS p, ",", ITOS q, ")-(", ITOS u, ",", ITOS v, ")  CofG (", ITOS x,
         ",", ITOS y, ")  Win (", ITOS a, ",", ITOS b, ")-(", ITOS c, ",", ITOS d, ")\n\n"],
        (a,b): e,
        (c,d): f,
        (e,f): Winrect s,
        (p,q): m,
        (u,v): n,
```

```
        (m,n): Picrect s,
        (x,y): CentOfGrav s;

:: Shstatus Lifestatus -> STRING;
   Shstatus DEAD -> "DEAD";
   Shstatus STABLE -> "STABLE";
   Shstatus DYNAMIC -> "DYNAMIC";
```

The `ShowRect` function prints the viewing window, by using a recursive loop through relative coordinates (x, y) ranging from $(0, 0)$ to $(w-1, h-1)$. The loop is started at $(-1, 0)$, with the negative coordinate used to signal the start of a line. When a coordinate (w, i) is reached, this signals the end of a line, and the scan continues with $(-1, i+1)$. When the coordinates of a point in the given list are reached, a star is printed at that position. An unexpected failure in the ordering of the points list (which has been sorted at this point) causes abortion, using `Concat` to assemble an error message string.

```
:: ShowRect INT INT PntList !UNQ FILE -> UNQ FILE;
   ShowRect w h l f -> ShowRectAux -1 0 w h l f',  == start at x=-1, y=0
                       f': DashLine w f;

:: ShowRectAux !INT !INT !INT !INT PntList !UNQ FILE -> UNQ FILE;
   ShowRectAux x y w h l f
        -> FWriteC '\n' (DashLine w f), IF >= y h  == end of rectangle
        -> ShowRectAux -1 (++ y) w h l (FWriteS "|\n" f), IF >= x w
        -> ShowRectAux 0 y w h l (FWriteS "| " f), IF < x 0;
   ShowRectAux x y w h [] f
        -> ShowRectAux (++ x) y w h [] (FWriteS Blank f);
   ShowRectAux x y w h l:[(x',y')|t] f
        -> p:ShowRectAux (++ x) y w h l (FWriteS Blank f), IF < y y'
        -> q:ABORT (Concat ["Error in ShowRectAux at (" , ITOS x', ",", ITOS y', ")\n"]), IF > y y'
        -> p, IF < x x'  == now have y=y'
        -> q, IF > x x'
        -> ShowRectAux (++ x) y w h t (FWriteS Star f);  == now have x=x' and y=y'

:: DashLine !INT !UNQ FILE -> UNQ FILE;
   DashLine i f -> FWriteS "---\n" f, IF = 0 i
                -> DashLine (-- i) (FWriteS "--" f);
```

The `Showstate` operation prints the state of the current pattern (held in the `Lifestate` variable) to a file, and also prints the viewing window if the pattern is not dead:

```
:: Showstate Lifestate !UNQ FILE -> UNQ FILE;
   Showstate s f
        -> f', IF Dead s
        -> ShowRect (Width s) (Height s) (Croppedpoints s) f',
           f': FWriteL (Statusline s) f;
```

The `Loop` operation performs the major user interaction. It uses `Readcommand` to read a user command, and calls `LoopAux` to perform the appropriate action, by pattern-matching. The parameters required are the current `Lifestate s`, the standard input-output file `f`, and the file system `fs`. The result is a modified file system, so the `LoopAux` function must take responsibility for closing the standard input/output file.

```
:: Loop !Lifestate !UNQ FILE !FILES -> !FILES;
   Loop s f fs -> LoopAux com s f'' fs,
                  f': ShortCommands f,
                  (com, f''): Readcommand f';
```

The `LoopAux` function pattern-matches on the various possible commands. It handles the `Quit` command by closing the standard input/output and terminating. The `Help` and `Illegal` commands result in the output of the messages defined above:

```
:: LoopAux Command !Lifestate !UNQ FILE !FILES -> !FILES;
   LoopAux Quit s f fs -> fs',
                          (dummy, fs'): FCloseStd f fs;
   LoopAux Help s f fs -> Loop s (DoHelp f) fs;
   LoopAux (Illegal mess) s f fs -> Loop s (DoIllegal mess f) fs;
```

The `Erase`, `GoToCentre`, `GoTo` and `ChgPnt` commands alter the current pattern using the appropriate abstract data type operations, then print the new state:

```
   LoopAux Erase s f fs -> Loop s' (Showstate s' f) fs,
                           s': Loadpoint [] s;
   LoopAux GoToCentre s f fs -> Loop s' (Showstate s' f) fs,
                                s': GoCentre s;
   LoopAux (GoTo pnt) s f fs -> Loop s' (Showstate s' f) fs,
                                s': CentreAt pnt s;
   LoopAux (ChgPnt pnt) s f fs -> Loop s' (Showstate s' f) fs,
                                  s': Changepoint ABSOLUTE pnt s;
```

The `Load`, `Store`, and `Pscript` commands handle input/output similarly to the *lifeconvert* program in section 7:

```
   LoopAux (Load name) s f fs
```

```
    -> Loop s (FWriteL ["CAN'T OPEN FILE '", name, "'\n"] f) fs', IF NOT ok
    -> Loop s (FWriteL ["NO CELLS FOUND IN FILE '", name, "'\n"] f) fs'', IF = 0 (Numcells s')
    -> Loop s' (Showstate s' f) fs'',
      (ok, g, fs'): FOpen name FReadText fs,
      (pnts, g'):  ReadPtsList g,
      s':  Loadpoint pnts s,
      (ok', fs''): FClose g' fs';

  LoopAux (Store name) s f fs
    -> Loop s (FWriteL ["CAN'T OPEN FILE '", name, "'\n"] f) fs', IF NOT ok
    -> Loop s (FWriteL ["CAN'T CLOSE FILE '", name, "'\n"] f) fs'', IF NOT ok'
    -> Loop s (FWriteS messg f) fs'',
       (ok, g, fs'): FOpen name FWriteText fs,
       g': WritePtsList (Allpoints s) g,
       (ok', fs''): FClose g' fs',
       messg: Concat ["Pattern written to file ", name, "\n"];

  LoopAux (Pscript name) s f fs
    -> Loop s (FWriteL ["CAN'T OPEN FILE '", name, "'\n"] f) fs', IF NOT ok
    -> Loop s (FWriteL ["CAN'T CLOSE FILE '", name, "'\n"] f) fs'', IF NOT ok'
    -> Loop s (FWriteS messg f) fs'',
       (ok, g, fs'): FOpen name FWriteText fs,
       g': WritePS head (Picrect s) (Allpoints s) g,
       head: Concat [name, "   ", ITOS (Numcells s), " cells   step ", ITOS (Stepno s)],
       (ok', fs''): FClose g' fs',
       messg: Concat ["PostScript pattern written to file ", name, "\n"];
```

The `Run` command prints a message if the existing pattern is already stable or dead, otherwise it executes the first step and calls `DoRun`. This function loops until the pattern becomes stable, or the desired number of steps is reached. It always prints the current viewing window before performing subsequent steps:

```
  LoopAux (Run track i) s f fs
    -> Loop s (FWriteS "\nPATTERN IS DEAD: NO CHANGE\n" f) fs, IF Dead s
    -> Loop s (FWriteS "\nPATTERN IS STABLE: NO CHANGE\n" f) fs, IF Stable s
    -> DoRun track (-- i) (Trackstep track s) f fs;

:: DoRun !TRACKMODE !INT Lifestate !UNQ FILE !FILES -> FILES;
   DoRun track i s f fs
    -> Loop s (FWriteS "DEAD\n" f) fs, IF Dead s
    -> Loop s (FWriteS "STABLE\n" f) fs, IF Stable s
    -> Loop s (FWriteS "STOPPED\n" (Showstate s f)) fs, IF = i 0
    -> DoRun track (-- i) (Trackstep track s) (Showstate s f) fs;
```

Finally the `Start` rule opens the file system and standard input/output file before calling the `Loop` function with an initially empty pattern. The file system is closed after `Loop` terminates:

```
:: Start UNQ WORLD -> UNQ WORLD;
   Start world
    -> world'',
       (file1, world'): OpenFiles world,
       (f, file2): StdIO file1,
       s: Emptystate W H,
       file3: Loop s (Showstate s f) file2,
       world'': CloseFiles file3 world';
```

11 Some Useful GUI Operations

Before we give the final GUI-based LIFE animation program, we provide a module, *winmisc*, which contains some useful GUI operations:

```
IMPLEMENTATION MODULE winmisc;
IMPORT lifemisc, deltaEventIO, deltaIOSystem;
FROM deltaDialog IMPORT Beep, OpenNotice;
FROM deltaPicture IMPORT DrawFunction, Point, Rectangle, SetFont, MovePenTo, DrawString;
FROM deltaFont IMPORT Font, FontName, FontStyle, FontSize, FontInfo, SelectFont, FontMetrics;
```

The following macro divides a number by two, using shifting:

```
MACRO
   Half x -> SHIFTR% x 1;
```

The `Message` function of section 8 appears here in a general form. Both user and system states are provided as (strict) arguments, but the function is generic in that it accepts any system state:

```
RULE
:: Message [STRING] !s !(IOState s) -> (!s, !IOState s);
   Message mess s io
```

```
    -> (s, io'),
       (dummybutton, io'): OpenNotice notice io,
       notice: Notice mess (NoticeButton 1 "OK") [];
```

We can use this function to define `Error` similarly to section 8, and `Crash` which handles fatal errors by quitting after the dialog box is dismissed:

```
:: Error STRING !s !(IOState s) -> (!s, !IOState s);
   Error str s io
     -> Message ["AN ERROR HAS OCCURRED", "", str] s (Beep io);

:: Crash STRING !s !(IOState s) -> (!s, !IOState s);
   Crash str s io
     -> (s', QuitIO io'),
        (s',io'): Message ["A FATAL ERROR HAS OCCURRED", "", str] s (Beep io);
```

The `Verify` function opens a dialog box containing a message, and applies the given state transition function only if the user selects the **Yes** (instead of the default **Cancel**) button:

```
:: Verify STRING (=> s (=> (IOState s) (s, IOState s))) !s !(IOState s) -> (!s, !IOState s);
   Verify mess f s io
     -> (s, io'), IF = button 1
     -> f s io',
        (button, io'): OpenNotice notice io,
        notice: Notice [mess] (NoticeButton 1 "Cancel") [NoticeButton 2 "Yes"];
```

The `ExtendToState` function extends a state transition function on the system state to act on a pair of user and system states:

```
:: ExtendToState !(=> (IOState s) (IOState s)) !s !(IOState s) -> (!s, !IOState s);
   ExtendToState f s io -> (s, f io);
```

The following two functions convert boolean values to the algebraic data types `MarkState` and `SelectState` used in GUI specifications:

```
:: BTOMarkState !BOOL -> MarkState;
   BTOMarkState  b -> Mark, IF b
                   -> NoMark;

:: BTOSelectState !BOOL -> SelectState;
   BTOSelectState  b -> Able, IF b
                     -> Unable;
```

We also provide a functions for drawing text in a window in any desired font. The following auxiliary function returns a font and its associated metrics, given the name, style (e.g. bold or italic), and size. The default font is returned if the desired font is not available. A legal font and its corresponding metrics are always returned. The calculated metrics are maximum ascent above baseline, maximum descent below baseline, maximum character width, and full height (including leading):

```
:: GetFont !FontName ![FontStyle] !FontSize  -> (!Font, !INT, !INT, !INT, !INT);
   GetFont fname styles siz
     -> (font, asc, desc, mw, height),
        (ok, font): SelectFont fname styles siz,
        (asc, desc, mw, lead): FontMetrics font,
        height: + asc (+ desc lead);
```

The `ShowText` function recursively draws a list of strings, with the baseline of the first line starting at (x, y). Subsequent lines start at $(x, y + h)$, $(x, y + 2h)$, etc. Strings which fall vertically outside the given rectangle are not drawn (i.e. we restrict $\texttt{lo} \leq y \leq \texttt{hi}$):

```
:: ShowText ![STRING] !INT !INT !INT !INT !INT !Rectangle !Picture -> Picture;
   ShowText text x y asc desc h ((a,b),(c,d)) pic
     -> ShowTextAux text x y h lo hi pic,
        lo: - b desc,
        hi: + d asc;

:: ShowTextAux ![STRING] !INT !INT !INT !INT !INT !Picture -> Picture;
   ShowTextAux [] x y h lo hi pic -> pic;
   ShowTextAux [s|t] x y h lo hi pic
     -> pic, IF > y hi
     -> ShowTextAux t x (+ y h) h lo hi pic, IF < y lo
     -> ShowTextAux t x (+ y h) h lo hi pic'',
        pic': MovePenTo (x,y) pic,
        pic'': DrawString s pic';
```

The `DrawText` function (which is exported) calls `ShowText` to draw text in a given rectangle, with the top left corner of the first character at the point (x, y). The requested font information is processed using `GetFont`. This function was used to generate the text in the dialog box of Figure 7.

```
:: DrawText !Rectangle !FontName ![FontStyle] !FontSize  !Point ![STRING] !Picture -> Picture;
   DrawText rect fname styles siz (x,y) text pic
```

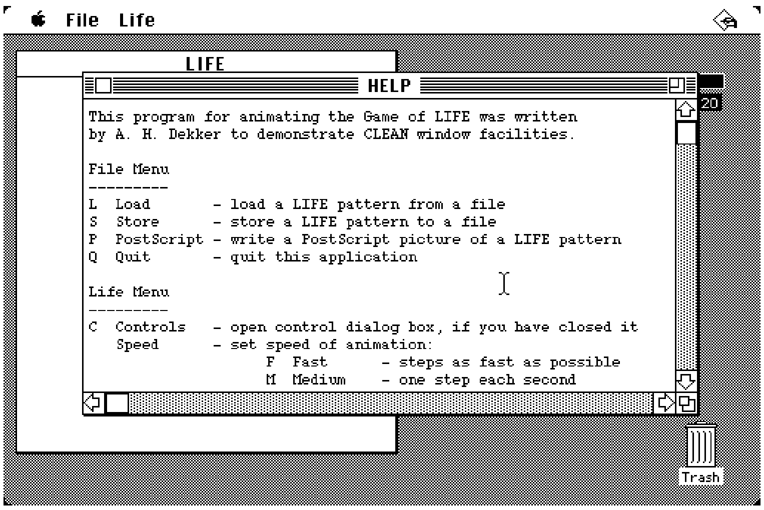

Figure 4: Scrollable text window on Macintosh

```
        -> ShowText text x (+ y asc) asc desc height rect pic',
           (font, asc, desc, mw, height): GetFont fname styles siz,
           pic': SetFont font pic;
```

The `TextWindow` function defines a scrollable window for viewing a given list of strings in a specified font. The function is generic for any user and system state (which must be specified to be unique). The window has a special 'I-Beam' cursor, but does not respond to keyboard or mouse events. An example window created by a call to this function is shown in Figure 4.

```
:: TextWindow WindowId WindowPos WindowTitle InitialWindowSize FontName [FontStyle] FontSize [STRING]
                                                                           -> WindowDef UNQ s UNQ io;
   TextWindow winid winpos wintitle initsize fname styles siz text
     -> ScrollWindow winid winpos wintitle
            (ScrollBar (Thumb 0) (Scroll hscroll))
            (ScrollBar (Thumb 0) (Scroll vscroll))
            windomain minsize initsize
            (TextUpdate font text x y asc desc h)
            [Cursor IBeamCursor],
```

The specification defines a domain whose height is just sufficient to hold the given strings, and whose width allows for 100 characters of maximal width (calculating an exact width is also possible, but more time-consuming). Clicking on the scrollbars scrolls in units of 10 characters horizontally, or 3 lines vertically. The first character is positioned so as to allow a blank margin of half a character width horizontally, and half a line vertically. An initial window size is specified as a parameter to the function, but once created the window can be resized up to the domain size, or down to a minimum size (which in this case allows 3 lines of 30 characters to be displayed):

```
           (font, asc, desc, mw, h): GetFont fname styles siz,
           length: Len text,
           windomain: ((0,0),(width,height)),
           height: * h (++ length),
           width: * mw 101,
           hscroll: * 10 mw,
           vscroll: * 3 h,
           minsize: (* 31 mw, * 4 h),
           x: Half mw,
           y: + asc (Half h);
```

All windows must have an update function, which takes a list of rectangles (`UpdateArea`), and a user state, and calculates a list of drawing functions which redraw the required parts of the window. We use `Map` and `ShowText` to do this. The user state `s` must be specified unique here, since the `IOState` context in most function declarations (which implicitly specifies `s` as unique) is not present:

```
:: TextUpdate Font [STRING] !INT !INT !INT !INT !INT UpdateArea !UNQ s -> (!s, [DrawFunction]);
   TextUpdate font text x y asc desc h areas s
     -> (s, [SetFont font | Map (ShowText text x y asc desc h) areas]);
```

The *winmisc* definition module (not shown) exports the required functions, and the types which they use.

12 A User-Defined Control

When the predefined GUI facilities (buttons, slider bars, etc.) provided by CLEAN are insufficient, it is possible to write a *control* facility whose look and feel are completely user-defined. A generic control can be specified similarly to the operations in the previous section. However, to separately compile a specification of a control specifically for LIFE animation, we require a header file specifying the types used by the GUI-based animation program:

```
IMPLEMENTATION MODULE winhead;
IMPORT lifestate, winmisc, lifeio;
```

The user state for this program contains a boolean flag indicating if the animation is currently running, the size (in pixels) of cells in the viewing window, a boolean flag indicating if tracking is used, the `Lifestate` abstract data type, and the file system. The last three of these were also used as function parameters to `DoRun` in the text-based animation of section 10.

```
TYPE
:: BlockSize -> INT;
:: RunState  -> BOOL;
:: UNQ State -> (!RunState, !BlockSize, !TRACKMODE, !Lifestate, !FILES);
:: UNQ IO    -> IOState State;
```

The definition module for the header (not shown) is identical to the implementation module.

The control which we define is used to reposition the viewing window around the LIFE pattern. It provides a reduced view of the rectangle occupied by the picture and the square occupied by the viewing window (the *look*) and also the ability to move the viewing window with the mouse (the *feel*).

A control in CLEAN cannot affect either the user or system states in the way that e.g. dialog boxes can. However, a control is always embedded inside a dialog box, and it can alter both the state of the dialog box in which it is placed, as well as its own internal *control state*. For this example, the control cannot cause the viewing window to actually move. However, it activates a button in its parent dialog box, and clicking this button causes the actual movement to take place.

```
IMPLEMENTATION MODULE wincontrol;
IMPORT winhead, deltaDialog, deltaPicture;
```

The `Scaletype` type is used to specify a translation and scaling of the viewing window to the control's own drawing domain. The size of this domain is specified by a macro, and the graphics defining the control's look and feel are automatically cropped by CLEAN to fit the domain. The control's internal state is an algebraic data type which provides a kind of dynamic typing. As a dummy initial state we use `IntCS 0`, while the state `PairCS` (`IntCS` x) (`IntCS` y) is used to specify that the centre of the viewing window should be moved to the coordinates (x, y).

Because of the complexity of this control, we have chosen a simple control state which alters in response to mouse events, and a look and feel which alter in response to changes in the LIFE pattern. It would also be possible to define this control using static look and feel functions, but this would require a much more complex control state to specify properties of the current pattern and viewing window. Programming the GUI interface is much easier if we allow the look and feel to change with time.

```
TYPE
:: ScaleType -> (!Pnt, !INT);
MACRO
   ControlSize -> 80;
   ControlDomain -> ((0,0),(ControlSize,ControlSize));
   CrossSize -> 4;
   InitCS -> IntCS 0;
```

The control is parameterised on the identity number and position it will have inside its parent dialog box, the current `RunState` and `Lifestate` which it will show, and the identity number of its associated dialog button. If the animation is running or the LIFE pattern is dead, the feel of the control is disabled:

```
RULE
:: TheControl DialogItemId ItemPos RunState Lifestate DialogItemId -> DialogItem State IO;
   TheControl id pos run life buttonid
     -> Control id pos ControlDomain ability InitCS look feel fn,
        ability: BTOSelectState (NOT (OR run (Dead life))),
        (look, feel): LookAndFeel life,
        fn: TheControlFn buttonid;
```

The look and feel of the control both depend on the `Lifestate` in similar ways, so are partly calculated by one function. If the LIFE pattern is dead, the look is specified by `DeadLook` (which simply fills the control domain with a light grey pattern), and the feel is ignored because it is disabled. Otherwise, the look and feel are specified by `TheControlLook` and `TheControlFeel`, which depend on the rectangle $((a, b), (c, d))$ occupied by the pattern, the size $(w \times h)$ of the viewing window, the 'centre of gravity' `cofg` of the pattern, the upper left corner `upleft` of the viewing window, and the scaling factor necessary:

```
:: LookAndFeel Lifestate -> (ControlLook, ControlFeel);
   LookAndFeel life
     -> (DeadLook, feel), IF Dead life
     -> (TheControlLook scale abcd upleft w h cofg, feel),
        feel: TheControlFeel scale upleft w h,
        abcd: Picrect life,
        (ab,cd): abcd,
        (a,b): ab,
        (c,d): cd,
        win: Winrect life,
        (upleft,botright): win,
```

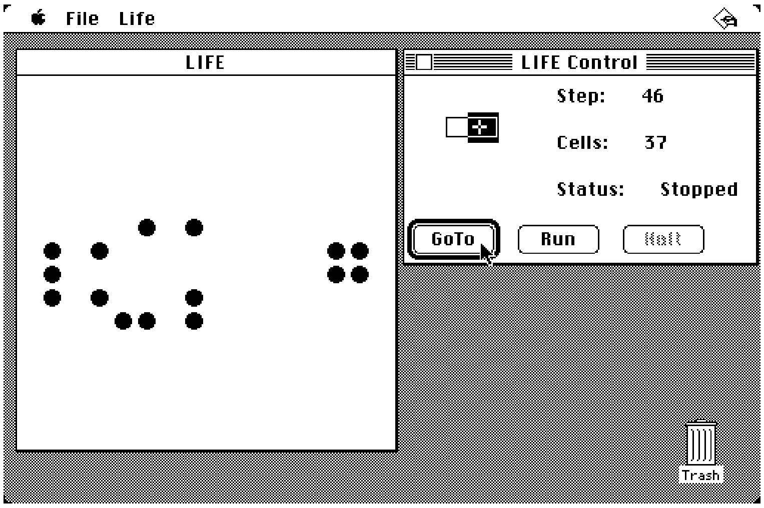

Figure 5: Active LIFE control on Macintosh

```
        x: + (* 2 w) (- c a),
        y: + (* 2 h) (- d b),
        scale:((- a w, - b h), Max  x y),
        w: Width life,
        h: Height life,
        cofg: CentOfGrav life;

:: DeadLook SelectState ControlState -> [DrawFunction];
   DeadLook ability anycs -> [SetPenPattern LtGreyPattern, FillRectangle ControlDomain];
```

The look is defined by a function which accepts (in addition to parameters supplied by `LookAndFeel`) the current selection ability of the control, and the current control state. If the control is active, the rectangle occupied by the pattern is drawn by `ControlRect`, the viewing window is drawn as a solid square by `ControlActive` or `ControlBox`, and the centre of gravity of the pattern is drawn as a cross by `ControlCross`. The `ControlActive` function is used when the position of the viewing window box has already been moved from its original position, and the `ControlBox` function when it has not. The translation between viewing window coordinates and control domain coordinates is done by `ControlScale` and `ControlUnScale`. An example of an active control (after dragging the black square representing the viewing window, and just before clicking the **GoTo** button) is shown in Figure 5. Clicking on the **GoTo** button causes the window to be moved to the new area of the pattern, and the button to become disabled. If the control is inactive, the square representing the current viewing window is shown in dark grey instead of black, as shown in Figure 6.

```
:: TheControlLook ScaleType Rect Pnt !WIDTH !HEIGHT Pnt !SelectState ControlState -> [DrawFunction];
   TheControlLook scale rect upleft w h cofg ability (PairCS (IntCS x) (IntCS y))
     -> [ControlRect scale rect, SetPenMode XorMode,
         ControlActive scale w h x y | ControlCross scale cofg];
   TheControlLook scale rect upleft w h cofg Unable notactivecs
     -> [ControlRect scale rect, SetPenMode XorMode, SetPenPattern DkGreyPattern,
         ControlBox scale upleft w h | ControlCross scale cofg];
   TheControlLook scale rect upleft w h cofg Able notactivecs
     -> [ControlRect scale rect, SetPenMode XorMode,
         ControlBox scale upleft w h | ControlCross scale cofg];

:: ControlRect ScaleType Rect -> DrawFunction;
   ControlRect scale ((a,b),(c,d))
     -> DrawLine (ControlScale scale a b, ControlScale scale c d), IF = a c || = b d
     -> DrawRectangle (ControlScale scale a b, ControlScale scale c d);

:: ControlActive ScaleType !WIDTH !HEIGHT !INT !INT -> DrawFunction;
   ControlActive scale w h p q
     -> FillRectangle (ControlScale scale a b, ControlScale scale c d),
        a: - p (Half w),
        b: - q (Half h),
        c: -- (+ a w),
        d: -- (+ b h);

:: ControlBox ScaleType Pnt !WIDTH !HEIGHT -> DrawFunction;
   ControlBox scale (a,b) w h
     -> FillRectangle (ControlScale scale a b, ControlScale scale c d),
        c: -- (+ a w),
```

```
        d: -- (+ b h);

:: ControlCross ScaleType Pnt -> [DrawFunction];
   ControlCross scale (x,y)
     -> [SetPenSize (2,2), DrawLine ((- x' CrossSize, y'), (+ x' CrossSize, y')),
         DrawLine (( x', - y' CrossSize), (x', + y' CrossSize)), SetPenNormal],
        (x',y'): ControlScale scale x y;

:: ControlScale ScaleType !INT !INT -> Point;
   ControlScale ((a,b),s) x y
     -> (/ (* (- x a) ControlSize) s, / (* (- y b) ControlSize) s);

:: ControlUnScale ScaleType !INT !INT -> Pnt;
   ControlUnScale ((a,b),s) x y
     -> (+ a x', + b y'),
        x': / (* x s) ControlSize,
        y': / (* y s) ControlSize;
```

The feel specifies the control's response to mouse events, and is defined by a function which accepts (in addition to parameters supplied by `LookAndFeel`) the mouse event and the current control state. The response to the mouse event is a new control state and a graphical change. This control reponds to all mouse events in the same way, by scaling the mouse position (a, b) and taking it as the desired centre for the viewing window. The previous box is erased by redrawing it in `XorMode`, and a box is drawn at the new position. The net effect is that the black square (representing the viewing window) follows the mouse as long as the mouse button is pressed. The control state is set to the new position with each mouse event. It is important to note that CLEAN always resets the pen to its defaults before applying a draw function list to a control, so the feel must set the pen mode on every event:

```
:: TheControlFeel ScaleType Pnt !WIDTH !HEIGHT MouseState ControlState -> (ControlState, [DrawFunction]);
   TheControlFeel scale upleft w h ((a,b),button,mods) cs:(PairCS (IntCS x) (IntCS y))
     -> (cs, []), IF = x x' && = y y'
     -> (PairCS (IntCS x') (IntCS y'),
         [SetPenMode XorMode, ControlActive scale w h x y, ControlActive scale w h x' y', SetPenNormal]),
        (x',y'): ControlUnScale scale a b;
   TheControlFeel scale upleft w h ((a,b),button,mods) notactivecs
     -> (PairCS (IntCS x) (IntCS y),
         [SetPenMode XorMode, ControlBox scale upleft w h, ControlActive scale w h x y, SetPenNormal]),
        (x,y): ControlUnScale scale a b;
```

In addition to the control feel, mouse events occurring inside the control domain also cause the execution of a state transition function on the parent dialog's own state. In this case, a specified button in the dialog is enabled (the **GoTo** button in Figure 5):

```
:: TheControlFn DialogItemId DialogInfo (DialogState State IO) -> DialogState State IO;
   TheControlFn buttonid dinfo dstate -> EnableDialogItems [buttonid] dstate;
```

We also specify in this module the method for the associated button in the parent dialog. Since the button is only enabled on a mouse event in the control, the control's internal state should have been set to contain a desired window position (x, y). The button alters the window position in the `Lifestate`, changes the parent dialog by disabling the button, and resets its look to reflect the new state (the actual visible window is not updated by this function, but by the calling program itself). The method function is parameterised on the identity numbers of the parent dialog, the control itself, and the associated button:

```
:: ControlApplyButton !DialogId !DialogItemId !ControlState !DialogItemId !State !IO -> (!State, !IO);
   ControlApplyButton thedialog id (PairCS (IntCS x) (IntCS y)) button s:(run,bsize,track,life,files) io
     -> (s',io'),
        life': CentreAt (x,y) life,
        s': (run, bsize, track, life', files),
        io': ChangeDialog thedialog [DisableDialogItems [button], ChangeTheControl id run life'] io;
   ControlApplyButton thedialog id notactivecs button s io
     -> Crash "GoTo button unexpectedly enabled" s io;
```

Finally the following function resets the control's look, feel, and internal state in response to a change in the `RunState` or `Lifestate`:

```
:: ChangeTheControl !DialogItemId !RunState !Lifestate !(DialogState State IO) -> DialogState State IO;
   ChangeTheControl id run life dstate
     -> DisableDialogItems [id] dstate''', IF run || Dead life
     -> EnableDialogItems [id] dstate''',
        dstate': ChangeControlState id InitCS dstate,
        dstate'': ChangeControlLook id look dstate',
        dstate''': ChangeControlFeel id feel dstate'',
        (look, feel): LookAndFeel life;
```

The control definition module (not shown) exports only the definitions of the control and its associated button and update function.

13 The Final GUI Animation Program

We are finally in a position to fit together the concepts from previous sections, and present a GUI-based animation program. It uses several of CLEAN's GUI facilities, as well as our user-defined control:

```
MODULE winanimate;
IMPORT winhead, wincontrol;
IMPORT deltaWindow, deltaFileSelect, deltaDialog, deltaPicture;
FROM deltaTimer IMPORT EnableTimer, DisableTimer, SetTimerInterval, TicksPerSecond;
FROM deltaMenu IMPORT EnableMenuItems, DisableMenuItems, MarkMenuItems, UnmarkMenuItems,
                      ChangeMenuItemTitles, ChangeMenuItemFunctions;
```

Macros specify the size of the viewing window, and initial values for various parameters:

```
MACRO
   WinSize      -> 256;
   WinDomain    -> ((0,0),(WinSize,WinSize));
   InitTrack    -> FALSE;
   InitSize     -> 16;
   InitInterval -> 1;
   InitRun      -> FALSE;
   InitLife     -> Emptystate height height,
                   height: / WinSize InitSize;
```

The `Start` rule is similar to that of section 8, but with several GUI facilities (two menus, a window, two dialog boxes, and a timer). The steps of the animation occur at each 'tick' of the timer. The initial `state` is created using initial parameter values and the initial file system. The final file system is extracted from the final state so that it can be closed. The `StartIO` function includes an action `InitAction` which is to be performed immediately after startup:

```
RULE
:: Start UNQ WORLD   -> UNQ WORLD;
   Start world
     -> world''',
        (filesystem, world'): OpenFiles world,
        (events, world''): OpenEvents world',
        state: (InitRun, InitSize, InitTrack, InitLife, filesystem),
        menu: MenuSystem [FileMenu, LifeMenu],
        window: WindowSystem [TheWindow],
        dialog: DialogSystem [HelpDialog, TheDialog InitRun InitLife],
        timer:  TimerSystem [TheTimer],
        (newstate, events'):  StartIO [menu, dialog, window, timer] state [InitAction] events,
        (run, bsize, track, life, filesystem'): newstate,
        world''': CloseEvents events' (CloseFiles filesystem' world'');
```

The initial startup action brings the viewing window to the front:

```
:: InitAction State IO -> (State, IO);
   InitAction s io -> (s, ActivateWindow TheWindowId io);
```

Each GUI facility we use will be defined to have an update function to bring it up-to-date. We define `UpdateAll` to bring the entire GUI system up-to-date:

```
:: UpdateTheMenus (State, IO) -> (State, IO);
   UpdateTheMenus pr -> UpdateFileMenu (UpdateLifeMenu pr);

:: UpdateAll (State, IO) -> (State, IO);
   UpdateAll pr -> UpdateTheMenus (UpdateTheWindow (UpdateTheDialog (UpdateTheTimer pr)));
```

We first present the definition of the **File** menu. Each GUI facility of the same type must have a unique numerical identifier, which is 1 for this menu (and 2 for the second menu). The **File** menu contains items to **Load**, **Store**, and save as **PostScript** a file, as well as **Quit**. These items all have indices beginning with 1, while those of the second menu all have indices starting with 2. It is often useful to macro-define these indices, but in this case they are only used in the menu definition and the update function immediately following:

```
:: FileMenu -> MenuDef State IO;
   FileMenu
     -> PullDownMenu 1 "File" Able [
            MenuItem 11 "Load"  (Key 'L') Able DoLoad,
            MenuItem 12 "Store" (Key 'S') Unable (DoWrite "Save as:" DoStore),
            MenuItem 13 "PostScript" (Key 'P') Unable (DoWrite "Save as PostScript:" DoPost),
            MenuSeparator,
            MenuItem 14 "Quit"  (Key 'Q') Able DoQuit
        ];
```

The update function for this menu disables any menu items inappropriate to the current pattern, and also disables file operations in a running animation. The disabled menu items are still visible, but cannot be selected. The use of an update function like this localises the menu item indices to one part of the program.

```
:: UpdateFileMenu (State, IO) -> (State, IO);
   UpdateFileMenu (s:(run,bsize,track,life,files), io)
```

```
        -> (s, DisableMenuItems [11,12,13] io), IF run
        -> (s, EnableMenuItems [11] (DisableMenuItems [12,13] io)), IF Dead life
        -> (s, EnableMenuItems [11,12,13] io);
```

The method for the **Load** item verifies the replacement of the current pattern (using the generic `Verify` operation defined in section 11), and then calls `DoLoadAux` which performs the file load similarly to section 8:

```
    :: DoLoad State IO -> (State, IO);
       DoLoad s:(run,bsize,track,life,files) io
         -> DoLoadAux s io, IF Dead life
         -> Verify "Do you really want to replace this pattern?" DoLoadAux s io;

    :: DoLoadAux State IO -> (State, IO);
       DoLoadAux s:(run,bsize,track,life,files) io
         -> (s', io'), IF NOT openselected
         -> Error (+S "Can't open file " input) s'' io', IF NOT ok
         -> Error (+S "No cells found in " input) s''' io', IF Dead life'
         -> UpdateAll (news,io'),
            (openselected, input, files', io'): SelectInputFile files io,
            s':(run, bsize, track, life, files'),
            (ok, g, files''): FOpen input FReadText files',
            s'':(run, bsize, track, life, files''),
            (points, g'): ReadPtsList g,
            life': Loadpoint points life,
            (dummy, files'''): FClose g' files'',
            s''':(run, bsize, track, life, files'''),
            news:(FALSE, bsize, track, life', files''');
```

The `DoWrite` function implements both **Store** and **PostScript**, being parameterised on the prompt to use in the output-file selection dialog box, and the specific file-write function (`DoStore` or `DoPost`):

```
    :: DoWrite STRING (=> STRING (=> Lifestate (=> !UNQ FILE UNQ FILE))) State IO -> (State, IO);
       DoWrite prompt writefn s:(run,bsize,track,life,files) io
         -> (s', io'), IF NOT saveselected
         -> Error (+S "Can't open file " output) s'' io', IF NOT okopen
         -> Error (+S "Can't close file " output) s''' io', IF NOT okclose
         -> (s''', io'),
            (saveselected, output, files', io'): SelectOutputFile prompt "" files io,
            s':(run, bsize, track, life, files'),
            (okopen, g, files''): FOpen output FWriteText files',
            s'':(run, bsize, track, life, files''),
            g': writefn output life g,
            (okclose, files'''): FClose g' files'',
            s''':(run, bsize, track, life, files''');

    :: DoStore STRING Lifestate !UNQ FILE -> UNQ FILE;
       DoStore fname life f -> WritePtsList (Allpoints life) f;

    :: DoPost STRING Lifestate !UNQ FILE -> UNQ FILE;
       DoPost fname life f
         -> WritePS head (Picrect life) (Allpoints life) f,
            head: Concat [fname, "   ", ITOS (Numcells life), " cells   step ", ITOS (Stepno life)];
```

The method for **Quit** is built out of two generic operations (`Verify` and `ExtendToState`) from section 11:

```
    :: DoQuit -> => State (=> IO (State, IO));
       DoQuit -> Verify "Do you really want to quit?" (ExtendToState QuitIO);
```

The second menu (with index 2) is the **Life** menu, which contains items to open the **Controls** dialog box, alter the **Speed** of the animation, alter the **Size** of cells in the viewing window, change the **Track** mode, **Erase** the entire pattern, and **Run** or stop the animation.

The **Speed** item is a submenu containing radio items (i.e. only one sub-item can be selected at any one time). This submenu specifies the time in seconds (0, 1, 5 or 10) between steps of the animation. It is convenient for the indices of the radio items to be computed from the time interval (as $2200 + t$), so that the radio item which is checked initially depends on the macro-specified initial time interval. The same strategy applies to the **Size** submenu:

```
    :: LifeMenu -> MenuDef State IO;
       LifeMenu
         -> PullDownMenu 2 "Life" Able [
                MenuItem  21  "Controls"    (Key 'C') Able DoControls,
                SubMenuItem  22  "Speed" Able [
                    MenuRadioItems  (+ 2200 InitInterval) [
                        MenuRadioItem 2200 "Fast" (Key 'F') Able (DoSpeed 0),
                        MenuRadioItem 2201 "Medium" (Key 'M') Able (DoSpeed TicksPerSecond),
                        MenuRadioItem 2205 "Slow" NoKey Able (DoSpeed (* 5 TicksPerSecond)),
                        MenuRadioItem 2210 "Very Slow" NoKey Able (DoSpeed (* 10 TicksPerSecond))
```

```
                    ]
                ],
                SubMenuItem  23  "Size" Able [
                    MenuRadioItems  (+ 2300 InitSize) [
                        MenuRadioItem  2304 "4x4" NoKey Able (DoSize 4),
                        MenuRadioItem  2308 "8x8" NoKey Able (DoSize 8),
                        MenuRadioItem  2316 "16x16" NoKey Able (DoSize 16),
                        MenuRadioItem  2332 "32x32" NoKey Able (DoSize 32)
                    ]
                ],
                CheckMenuItem 24 "Track" (Key 'T') Able (BTOMarkState InitTrack) DoTrack,
                MenuItem 25 "Erase" (Key 'E') Unable
                                (Verify "Do you really want to erase all cells?" DoErase),
                MenuItem 26 "Run" (Key 'R') Unable DoRun
            ];
```

The update function for the **Life** menu reflects the fact that a dead pattern cannot be run or erased, and a stable pattern cannot be run. More importantly, when the animation is running, the **Run** menu item is altered to **Halt**, with a corresponding change to the method function:

```
:: UpdateLifeMenu (State, IO) -> (State, IO);
   UpdateLifeMenu (s:(run,bsize,track,life,files), io)
     -> (s, DisableMenuItems [25] (EnableMenuItems [26]
             (ChangeMenuItemTitles [(26,"Halt")] (ChangeMenuItemFunctions [(26,DoHalt)] io)))), IF run
     -> (s, DisableMenuItems [25, 26] haltedio), IF Dead life
     -> (s, EnableMenuItems [25] (DisableMenuItems [26] haltedio)), IF Stable life
     -> (s, EnableMenuItems [25, 26] haltedio),
        haltedio: ChangeMenuItemTitles [(26,"Run")] (ChangeMenuItemFunctions [(26,DoRun)] io);
```

The **Controls** menu item opens the main dialog box (if it is not already open). The predefined `GetDialogInfo` function is used to indicate whether the dialog box is already open:

```
:: DoControls State IO -> (State, IO);
   DoControls s:(run,bsize,track,life,files) io
     -> (s, Beep io'), IF isopen
     -> (s, OpenDialog (TheDialog run life) io'),
        (isopen, dummyinf, io'): GetDialogInfo TheDialogId io;
```

The radio items in the **Speed** submenu alter the interval between 'ticks' of the timer. Those in the **Size** submenu alter the cell size in the viewing window, requiring update of the user state. The main dialog box and window are then updated to match:

```
:: DoSpeed !INT State IO -> (State, IO);
   DoSpeed interval s io -> (s, SetTimerInterval TheTimerId interval io);

:: DoSize !BlockSize State IO -> (State, IO);
   DoSize newsiz s:(run,bsize,track,life,files) io
     -> UpdateTheWindow (UpdateTheDialog (s',io)),
        height: / WinSize newsiz,
        life': Resize height height life,
        s': (run, newsiz, track, life', files);
```

The **Track** menu item toggles the tracking mode, which is recorded in the user state. The check-mark beside it in the menu is then adjusted to match. Unlike the marks beside radio items, those in a `CheckMenuItem` must be set and cleared explicitly:

```
:: DoTrack State IO -> (State, IO);
   DoTrack s:(run,bsize,track,life,files) io
     -> ((run,bsize,FALSE,life,files), UnmarkMenuItems [24] io), IF track
     -> ((run,bsize,TRUE,life,files), MarkMenuItems [24] io);
```

The **Erase** menu item stops the animation and and erases the current pattern. This requires updating the entire GUI system. The definition of the **Life** menu above only calls `DoErase` if the users confirms a verification notice.

```
:: DoErase State IO -> (State, IO);
   DoErase s:(run,bsize,track,life,files) io
     -> UpdateAll (s',io),
        s': (FALSE, bsize, track, Loadpoint [] life, files);
```

The last item in the menu calls either `DoRun` or `DoHalt`, depending on whether the animation is halted or running. The `DoRun` method enables the timer, disables mouse events in the viewing window (indicating this by altering the cursor icon locally within the window), sets the running flag in the user state, and updates the menus and main dialog box to match. The `DoHalt` method reverses all these changes. Figure 6 shows the viewing window and main dialog box while the animation is running.

```
:: DoRun State IO -> (State, IO);
   DoRun s:(run,bsize,track,life,files) io
     -> UpdateTheMenus (UpdateTheDialog (s',io')),
        s': (TRUE, bsize, track, life, files),
        io': DisableMouse TheWindowId
```

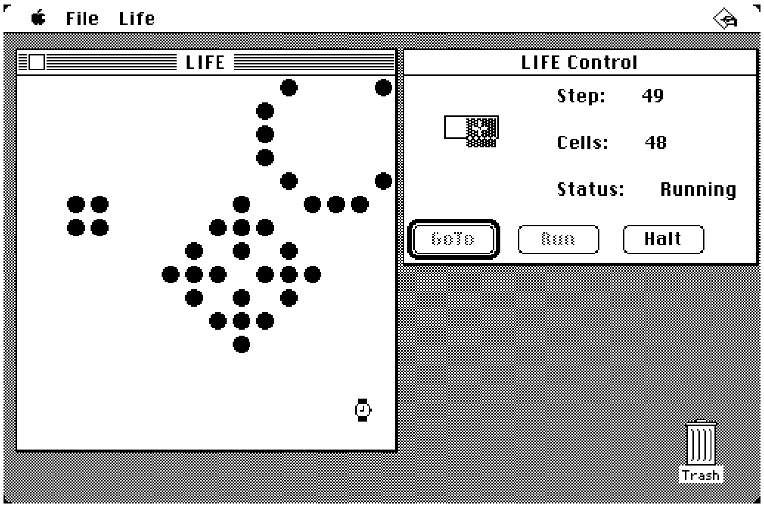

Figure 6: LIFE window for running animation on Macintosh

```
                    (ChangeWindowCursor TheWindowId BusyCursor (EnableTimer TheTimerId io));

:: DoHalt State IO -> (State, IO);
   DoHalt s:(run,bsize,track,life,files) io
     -> UpdateTheMenus (UpdateTheDialog (s',io')),
        s': (FALSE, bsize, track, life, files),
        io': EnableMouse TheWindowId
               (ChangeWindowCursor TheWindowId CrossCursor (DisableTimer TheTimerId io));
```

The animation is controlled by a *timer* device. Different timers must have different indices, but may have the same index as menus, windows, etc. (and similarly for menu, dialog-box, and window indices). The index is macro-defined to permit easy enabling and disabling of the timer elsewhere in the program:

```
MACRO
   TheTimerId -> 1;

RULE
:: TheTimer -> TimerDef State IO;
   TheTimer -> Timer TheTimerId Unable (* InitInterval TicksPerSecond) ClockFn;
```

The update function for the timer disables the timer if the animation has stopped. The `ClockFn` method (which is called on each 'tick' of the timer) performs one animation step and updates the entire GUI interface to match the result:

```
:: UpdateTheTimer (State, IO) -> (State, IO);
   UpdateTheTimer (s:(run,bsize,track,life,files), io)
     -> (s, EnableTimer TheTimerId io), IF run
     -> (s, DisableTimer TheTimerId io);

:: ClockFn TimerState State IO -> (State, IO);
   ClockFn tstate s:(run,bsize,track,life,files) io
     -> UpdateAll (s',io),
        life': Trackstep track life,
        run': NOT (Stable life'),
        s': (run', bsize, track, life', files);
```

The main dialog box is a command dialog (the most general case) which is opened at a specific position on the screen with default button 108. It contains the user-defined control described in section 12, and information on the current pattern (in *dynamic text* entries which can be changed by appropriate functions). The dialog box is shown in Figure 5. There are three dialog buttons: a default **GoTo** button associated with the control, and **Run** and **Halt** buttons which start and stop the animation in the same way as the corresponding menu items. The definition of the dialog box is parameterised on the current `RunState` and `LifeState` at the time it is opened, and this is used to determine which button should be initially enabled:

```
MACRO
   TheDialogId -> 1;

RULE
:: TheDialog RunState Lifestate -> DialogDef State IO;
   TheDialog run life
     -> CommandDialog TheDialogId "LIFE Control"
          [DialogPos (Pixel 270) (Pixel 10),   StandByDialog,   DialogMargin (Pixel 6) (Pixel 6)]
          108  == default button
          [TheControl 101 Left run life 108,
```

```
            StaticText 102 (RightTo 101) "Step: ",
            DynamicText 103 (RightTo 102) (Pixel 60) (ITOS (Stepno life)),
            StaticText 104 (Below 102) "Cells: ",
            DynamicText 105 (RightTo 104) (Pixel 60) (ITOS (Numcells life)),
            StaticText 106 (Below 104) "Status: ",
            DynamicText 107 (RightTo 106) (Pixel 60) (ShStatus run life),
            DialogButton 108 (Below 101) "GoTo" Unable GoToButton,
            DialogButton 109 (RightTo 108) "Run" (BTOSelectState (NOT (OR run (Stable life)))) RunButton,
            DialogButton 110 (RightTo 109) "Halt" (BTOSelectState run) HaltButton
           ];

:: ShStatus RunState Lifestate -> STRING;
   ShStatus TRUE life -> "Running";
   ShStatus FALSE life -> "Dead", IF Dead life
                       -> "Stable", IF Stable life
                       -> "Stopped";
```

The update function uses `ChangeDialog` and a list of change functions to update the dynamic text entries, the control, and the buttons:

```
:: UpdateTheDialog (State, IO) -> (State, IO);
   UpdateTheDialog (s:(run,bsize,track,life,files), io)
     -> (s,io'), IF NOT isopen
     -> (s,ChangeDialog TheDialogId [d1,d2,d3,d4,d5,drun] io'), IF run
     -> (s,ChangeDialog TheDialogId [d1,d2,d3,d4,d5] io'), IF Stable life
     -> (s,ChangeDialog TheDialogId [d1,d2,d3,d4,d5,dstop] io'),
        (isopen, dummyinf, io'): GetDialogInfo TheDialogId io,
        d1: ChangeDynamicText 103 (ITOS (Stepno life)),
        d2: ChangeDynamicText 105 (ITOS (Numcells life)),
        d3: ChangeDynamicText 107 (ShStatus run life),
        d4: DisableDialogItems [108,109,110],
        d5: ChangeTheControl 101 run life,
        drun: EnableDialogItems [110],
        dstop: EnableDialogItems [109];
```

The methods for the **Run** and **Halt** buttons are based on the corresponding menu items, while that of the **GoTo** button is based on the `ControlApplyButton` function of section 12. The control state and appropriate button indices are fed to this function, and the window is updated to reflect the changes to window position it makes in the `LifeState`:

```
:: RunButton DialogInfo State IO -> (State, IO);
   RunButton dinfo s io -> DoRun s io;

:: HaltButton DialogInfo State IO -> (State, IO);
   HaltButton dinfo  s io -> DoHalt s io;

:: GoToButton DialogInfo State IO -> (State, IO);
   GoToButton dinfo s io
     -> UpdateTheWindow (ControlApplyButton TheDialogId 101 cs 108 s io),
        cs: GetControlState 101 dinfo;
```

The viewing window allows for mouse but not keyboard events. Clicking the window's 'go-away' box has the same action as the **Quit** menu item. The *standby* facility ensures that clicking on an inactive window not only activates the window, but performs the usual action as well. This feature ensures that, although the window and dialog box cannot be active at the same time, they behave as if they can. Within the window, the cursor is by default a cross instead of its usual shape (although this is altered when the animation is running):

```
MACRO
   TheWindowId  -> 1;
   TheWindowPos -> (5,5);

RULE
:: TheWindow -> WindowDef State IO;
   TheWindow
     -> FixedWindow TheWindowId TheWindowPos "LIFE" WinDomain LifeUpdate
           [Mouse Able LifeMouse,   GoAway DoQuit,   Cursor CrossCursor,   StandByWindow];
```

The update function redraws the entire window, adjusts the cursor, and enables or disables mouse events:

```
:: UpdateTheWindow (State, IO) -> (State, IO);
   UpdateTheWindow (s:(run,bsize,track,life,files), io)
     -> (s', ChangeWindowCursor TheWindowId BusyCursor (DisableMouse TheWindowId io')), IF run
     -> (s', ChangeWindowCursor TheWindowId CrossCursor (EnableMouse TheWindowId io')),
        (s', drawlist): LifeUpdate [WinDomain] s,
        io': DrawInWindow TheWindowId [EraseRectangle WinDomain | drawlist] io;
```

Every window definition must contain a function like `LifeUpdate` which takes an update area (a list of rectangles) and a user state, and computes a list of draw functions. If portions of a window are obscured and then uncovered, CLEAN

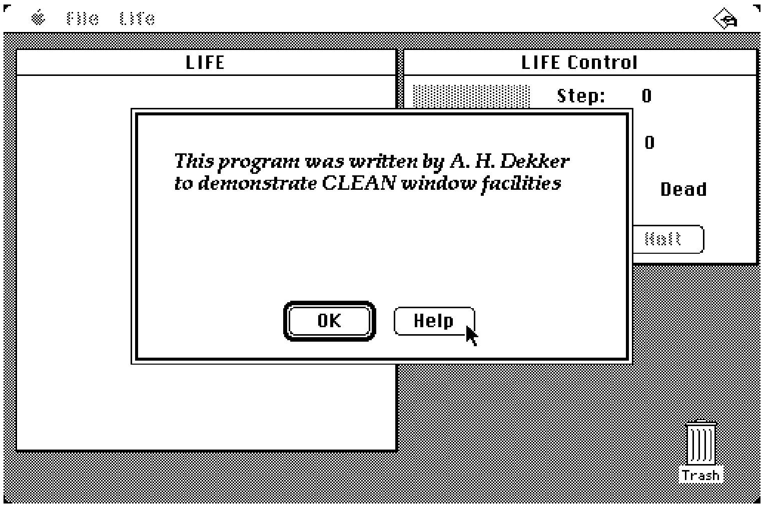

Figure 7: ABOUT dialog box on Macintosh

erases the affected rectangles and applies the draw functions computed by this function (cropping to fit the rectangles). The `LifeUpdate` function can also be used explicitly when we wish to redraw the window (as in `UpdateTheWindow` above). The draw functions returned by `LifeUpdate` completely ignore the list of rectangles, and draw a circle for each live cell in the viewing window:

```
:: LifeUpdate UpdateArea State -> (State, [DrawFunction]);
   LifeUpdate area s:(run,bsize,track,life,files)
     -> (s, Map (DrawCell bsize) (Croppedpoints life));

:: CellCircle BlockSize Pnt -> Circle;
   CellCircle bsize (x,y)
     -> ((x',y'),radius),
        radius: Max 1 (/ (* bsize 8) 20),
        half: Half bsize,
        x': + (* x bsize) half,
        y': + (* y bsize) half;

:: DrawCell BlockSize Pnt -> DrawFunction;
   DrawCell bsize pnt -> FillCircle (CellCircle bsize pnt);
```

The method for mouse events is `LifeMouse`. When the mouse button is clicked inside the window, the state of the selected cell is toggled (and it is inverted on the screen). To avoid inverting twice, it is important that there be no response to `ButtonStillDown` or `ButtonUp` events.

```
:: LifeMouse MouseState State IO -> (State, IO);
   LifeMouse ((x,y), ButtonDown, mods) s:(run,bsize,track,life,files) io
     -> UpdateTheMenus (UpdateTheDialog (s', io')),
        life': Changepoint RELATIVE (x',y') life,
        s': (run, bsize, track, life', files),
        io': DrawInActiveWindow
                [InvertCircle (CellCircle bsize (x',y'))] io,
        x': / x bsize,
        y': / y bsize;
   LifeMouse othermouse s io -> (s, io);
```

Finally CLEAN allows each application to have a special *about-dialog*, which is accessed in a system-dependent way (on the Macintosh it is selected from the Apple menu). This is a simple dialog box in which any desired graphics can be drawn, with an automatic **OK** button, and possibly also a **Help** button. It has no numerical index. In our case we use the `DrawText` function from section 11 to provide the contents of the dialog box, which is shown in Figure 7.

```
MACRO
   HelpDomain -> ((0,0),(300,100));

RULE
:: HelpDialog -> DialogDef State IO;
   HelpDialog
     -> AboutDialog "LIFE" HelpDomain
            [DrawText HelpDomain "Palatino" ["Italic","Bold"] 14 (10,10)
                ["This program was written by A. H. Dekker",
                 "to demonstrate CLEAN window facilities"]
            ]
            (AboutHelp "Help" DoHelp);
```

The method for the **Help** button opens a help file, reads it, and displays the contents in a text window (using the `TextWindow` definition from section 11). The resulting window is shown in Figure 4.

```
MACRO
   HelpFileName -> "life.help";
   HelpWindowId -> 2;
   HelpWindowPos -> (50,20);
   HelpWindowSize -> (400,200);

RULE
:: DoHelp State IO -> (State, IO);
   DoHelp s:(run,bsize,track,life,files) io
     -> Error (+S "Can't open help file " HelpFileName) s' io, IF NOT ok
     -> (s'', OpenWindows [helpwin] io),
        (ok, g, files'): FOpen HelpFileName FReadText files,
        s':(run, bsize, track, life, files'),
        (text, g'): ReadText g,
        (dummy, files''): FClose g' files',
        s'':(run, bsize, track, life, files''),
        helpwin: TextWindow HelpWindowId HelpWindowPos "HELP" HelpWindowSize "Courier" [] 10 text;

:: ReadText !UNQ FILE -> ([STRING], !UNQ FILE);
   ReadText f
     -> ([], f), IF SFEnd f
     -> ([s|rest], f''),
        (s, f'): FReadLine f,
        (rest,f''): ReadText f';
```

14 Conclusion

We have presented a stepwise development of a modular program for animating Conway's Game of Life, using CLEAN's GUI facilities. The major features of CLEAN have been introduced progressively during this development. Once accustomed to the novel style, programming in CLEAN is more rapid and error-free than in imperative languages. In particular, the elegant *unique type* mechanism ensures single-threading of input/output operations in a flexible way. Rewrite rules on constructors (as in section 10) help to simplify error-handling and thus make it easier to write single-threaded code. A disadvantage of CLEAN is that it only contains features common to both the Macintosh and Unix/X environments. As a result, there are no facilities for printing directly from within an application, for accessing Unix command-line arguments, or for accessing Macintosh resource forks. Some annoying but minor syntactic limitations of CLEAN derive from its earlier use as an intermediate language, and are likely to be resolved in future releases.

CLEAN's GUI facilities are somewhat less general than X or Motif. In particular, CLEAN inherits the Macintosh division between windows and dialog boxes. A window cannot contain buttons, although it is possible to mimic the effect by special handling of mouse events inside particular areas of the window. A dialog box cannot contain windows or other dialog boxes as components, although it may contain a user-defined *control* (which provides some of the facilities of a window, as we saw in section 12). However, the GUI features such as windows, menus, etc. provided by CLEAN have the advantage of cross-platform portability, and are very easy to define using list and algebraic data type (tree) structures. The definitions are first-class objects, so that we can write a function such as `TextWindow` (section 11) which returns a window definition. The method functions which respond to events are of course also first-class objects, and can be computed as was done with the look and feel functions in section 12. The CLEAN language provides the ability to rapidly develop readable and correct programs, and so should become increasingly popular.

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, second edition, 1990.

[2] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[3] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Graph Rewriting.* Addison-Wesley, 1993.

[4] A. Scedrov. A brief guide to linear logic. *Bulletin of the European Association for Theoretical Computer Science*, (41):154–165, June 1990.

[5] Marko van Eekelen, Halbe Huitema, Eric Nöcker, Sjaak Smetsers, and Rinus Plasmeijer. *Concurrent Clean language manual, version 0.8.4*, April 1993.