Concurrent Clean 1

Concurrent Clean

Version 0.8.4, April 1993

Concurrent Clean (Brus et al. (1987), Nöcker et al (1991), Plasmeijer & van Eekelen (1993)) is an experimental *purely lazy functional language* suited for evaluation on different kinds of machine architectures, varying from *simple computers* (like the Apple Macintosh) to *highly parallel architectures* (such as a Parsytec Supercluster).

The most important features of Concurrent Clean are:

- it is a *lazy* and *purely functional higher order* programming language;
- it is a *strongly typed* language (based on the well-known Milner/Mycroft type inferencing scheme (Milner (1978), Mycroft (1984))) including *polymorphic types*, *abstract types*, *algebraic types* and *synonym types*, as well as basic types; Also lists and tuples are predefined;
- it has a *module structure* with *implementation modules* and *definition modules* including a facility to implicitly and explicitly import definitions from other modules; it includes predefined modules (libraries) for basic operations (delta rules) on objects of basic types (*integers*, *reals*, *characters*, *booleans*, *strings* and *files*)
- *sharing of computation* can be expressed explicitly by the programmer in a general way including the possibility to specify cyclic structures;
- *modern input/output handling* (including *menu's*, *dialogues*, *mouse* and *window handling*) can be specified compactly and elegantly on a very high level of abstraction (Achten (1992)).
- *annotations* can be added to a function definition. With these annotations the evaluation order can be controlled by the programmer: functions can be made (partially) *eager* instead of lazy. When a function is known to be strict in a certain argument, the argument can be evaluated before the function is called. Such functions are in general more efficient. Further speed-ups can be obtained by defining *(partially) strict data structures* (Nöcker & Smetsers (1990)).
- it is also possible to split up the execution of the program in parts that are to be evaluated *interleaved* or in *parallel*. Processes can be created dynamically with arbitrary process topologies (for instance cyclic structures). The communication between processes does not have to be explicitly defined but is handled automatically.

Concurrent Clean has originally been designed to be used as an intermediate language between arbitrary (eager and lazy) functional programming languages and arbitrary sequential or parallel machine architectures. It is designed as a rather simple functional language to make it possible to concentrate on the implementation of the basic aspects only, instead of getting lost in the details of a specific full-featured functional language.

But, Concurrent Clean can also be used as an ordinary functional programming language in which (possibly parallel) computations are expressed in terms of graph rewriting. In contrast with most other functional programming languages Concurrent Clean supports only the most basic aspects of functional programming. Syntactic sugar (such as infix notations or complex list expressions) is mainly excluded.

Currently Concurrent Clean can be used as a (simple) programming language in its own right and as intermediate language between Miranda* and sequential architectures like Sun3, Sun4 and Macintosh as well as parallel architectures like Transputer based systems. In these systems Concurrent Clean itself is not translated directly to concrete machine code but to the machine code of an abstract machine called the Parallel ABC-machine. This abstract code is either interpreted by a software PABC-machine or compiled to concrete machine code of the host machine.

---

* Miranda™ is a trademark of Research Software Ltd.

The Concurrent Clean System

The Concurrent Clean System is the programming environment for Concurrent Clean. Currently the Concurrent Clean System contains

- a Concurrent Clean to PABC compiler (Smetsers (1989));
- a PABC simulator (Nöcker (1989)), such that (parallel) programs can be interpreted; it includes facilities for tracing, debugging and statistics;
- code generators that translate PABC code to native machine code (Groningen (1990,1991), Smetsers (1991), Kesseler (1990,1991)). At this moment versions for both Sun3 and all Macintosh machines (Motorola 680x0) as well as Sun4 (Sparc) are available. Furthermore a code generator for a Parsytec Supercluster transputer rack is available;
- a user-friendly Macintosh environment including a project manager with an automatic make facility and an editor (Wichers Schreur (1989), Huitema & Plasmeijer (1992));
- an automatic strictness analyser (Nöcker (1990,1992));
- a high level I/O interface with the Macintosh toolbox and with the X Window System using the Open Look™ Interface Toolkit (Achten (1992,1993)).

In the near future the Concurrent Clean System will provide:

- a code generator for Macintosh's and Suns communicating via a local area network;
- a facility to convert Miranda scripts to Concurrent Clean programs;
- the possibility to define and manipulate arrays and records (structures);
- further improvements to the efficiency of the sequential code;
- an even more user-friendly programming environment for both Macintosh as well as X Windows / Open Look systems. This environment will be programmed in Concurrent Clean.

For the final version of the Concurrent Clean System the following extensions are planned:

- more syntactical sugar will be added (infix expressions, overloading);
- an extension of the type system with a decidable subset of intersection typing (van Bakel (1989, 1990, 1993));
- high level process annotations that allow the programmer to express parallelism on a higher level;
- a tracer/debugger for Concurrent Clean;
- an implementation of the Concurrent Clean System in Concurrent Clean.

Differences with older versions

The description of Concurrent Clean given in this manual corresponds with the Concurrent Clean System version 0.8.4. The main differences with Concurrent Clean version 0.8.0, 0.8.1 and 0.8.3 are:

- The compiler, code generator, linker and run-time system have been improved.
- The I/O interface has been revised completely. It now contains a full implementation of dialogs and has become available for X Windows / Open Look systems (Sun4).

A more detailed description of the differences between Concurrent Clean version 0.8.4 and older versions can be found in chapter 7 of this manual. Since Concurrent Clean is an experimental language, changes and extensions may be expected in future versions of the language.

1 Some introductory examples

In this section we briefly introduce the flavour of Concurrent Clean by showing how some well-known and less well-known functional programs are written down in this formalism. We assume that the reader is familiar with other functional languages.

1.1 A simple Concurrent Clean program

```
MODULE Fac;

IMPORT delta;

MACRO
    Startvalue    ->  20;

RULE
::  Fac INT       ->  INT;
    Fac 0         ->  1;
    Fac n         ->  * n (Fac (-- n));

::  Start         ->  INT;
    Start         ->  Fac Startvalue;
```

A Concurrent Clean program is composed of modules. Modules are hierarchical. The top-most module is the main module. In the main module a `Start` rule should be declared of which the left-hand-side consists of the symbol `Start` and the right-hand-side corresponds with the initial expression (of any type) to be computed. Evaluation of a Concurrent Clean program will start with the evaluation of the function application defined in the right-hand-side of the `Start` rule. The `Start` rule may have one optional argument (of type `UNQ WORLD`), which is the environment parameter that is necessary to perform I/O actions (see section 1.8).

With the `IMPORT` statement all predefined arithmetic functions (delta rules) and related types are imported. `--` (integer decrement) and `*` (integer multiplication) are examples of such predefined functions of the predefined type `INT`.

It is possible to define macro's in Concurrent Clean. The right-hand-side of the macro definition will be substituted for every occurrence of the left-hand-side of the macro definition at compile time. Parameterized macro definitions are also possible.

Concurrent Clean expressions are generally written in prefix notation.

Rules starting with `::` are either new type definitions or type specifications of rewrite rules. In the latter case the type of the corresponding function is specified. In a Concurrent Clean program all rules for a certain function are called the alternatives for that function. It is required that the alternatives are grouped together, separated by semi-colons. The rule alternatives of a function definition are applied in textual order.

1.2 Lists and guards

The following Concurrent Clean program calculates prime numbers using the Sieve of Eratosthenes.

```
MODULE Eratosthenes;

==      Eratosthenes' Sieve

IMPORT delta;

RULE
```

```
::  Start               ->  [INT];
    Start               ->  Sieve (Gen 2);

::  Sieve [INT]         ->  [INT];
    Sieve [pr | str]    ->  [pr | Sieve (Filter pr str)];

::  Filter INT [INT]    ->  [INT];
    Filter pr [n | str] ->  Filter pr str , IF = (% n pr) 0
                        ->  [pr | Filter pr str];

::  Gen INT             ->  [INT];
    Gen n               ->  [n | Gen (++ n)];
```

In Concurrent Clean a comment line can be specified by preceding the line with '=='. Another way of specifying comment is enclosing it between comment brackets '<<' and '>>'. These comment brackets can enclose any number of characters or lines in a Clean program and can be nested.

Square brackets are used for denoting lists: `[]` is an empty list, `[a,b,c]` a list containing the three elements `a`, `b` and `c` and `[a | f]` denotes a list consisting of a list `f` prefixed with an element `a`. It is also possible to prefix a list with more than one element. For example: `[a, b, c | f]` denotes a list `f` prefixed by the elements `a`, `b` and `c`.

Concurrent Clean is a lazy language, so infinite lists (the result of `Gen 2` is the increasing list of all natural numbers starting with `2`) are evaluated lazily.

In Concurrent Clean conditional expressions are specified by means of guards (see the definition of `Filter`). Guards should be considered as an extension of the notion of patterns: an alternative only matches when its pattern matches and its guard evaluates to True. A group of alternatives with the same left-hand-side (pattern) is considered as one alternative consisting of one or more guarded expressions. Guards are separated from the rest of a guarded expression by an optional comma and the keyword `IF`. A guard is a boolean expression in which the infix operators `||` (conditional or) and `&&` (conditional and) can be used.

1.3 Higher-order functions

```
MODULE Map;

IMPORT delta;

RULE
::  Square INT          ->  INT;
    Square x            ->  * x x;

::  Map (=> x y) [x]    ->  [y];
    Map f []            ->  [];
    Map f [a | b]       ->  [f a | Map f b];

::  Start               ->  [INT];
    Start               ->  Map Square [42,43,44];
```

Higher order functions can be used freely. There is no difference between the use of full and partial (curried) applications of functions.

Types of higher order functions are specified using => (prefix notation) which corresponds to -> (infix notation) in languages like Miranda.

1.4 Strictness annotations

```
MODULE StrictNfib;

FROM deltaI IMPORT <, +, --, -;

RULE
```

```
::  Nfib !INT   ->  INT;
    Nfib n      ->  1 , IF < n 2
                ->  + (Nfib (-- n)) (Nfib (- n 2));

::  Start       ->  INT;
    Start       ->  Nfib 20;
```

Strictness annotations (!) can be used to indicate that a function is strict in a certain argument instead of lazy. Such an argument will be evaluated before the corresponding function is called. Generally, more efficient code can be generated for functions with strict arguments of basic types. Only arguments of which the forced evaluation cannot influence the termination behaviour of the program may be annotated as being strict. The strictness analyser of the Concurrent Clean System is able to derive the strictness information for many cases, such as for the example above. So generally there is no need to add annotations by hand.

In the example above the predefined delta rules are explicitly imported instead of implicitly as in the previous examples.

1.5 Strict Tuples

```
MODULE Complex;

IMPORT deltaR;

TYPE
::  Complex                 -> (!REAL, !REAL);

RULE
::  +C !Complex !Complex  ->  Complex;
    +C (r1,i1) (r2,i2)    ->  (+R r1 r2, +R i1 i2);

::  *C !Complex !Complex  ->  Complex;
    *C (r1,i1) (r2,i2)    ->  (-R (*R r1 r2) (*R i1 i2), +R (*R r1 i2) (*R r2 i1));
```

Strictness annotations can also be used in tuple types. These annotations indicate that the corresponding parts of the tuples can be evaluated immediately when the tuples are used. For instance, in case of the above-standing definition of the addition function `+C` for complex numbers this implies that the sums of the real and imaginary parts are computed directly when this function is called. Such strictness information cannot be derived by the Concurrent Clean System.

1.6 A cyclic example

In Concurrent Clean it is possible to explicitly express sharing of computation including the possibility to specify cyclic structures. This last possibility has been used in this example, a solution to the Hamming problem: calculate the (infinite) list of numbers having only 2, 3 and 5 as prime factors.

```
MODULE hamming;

IMPORT deltaI;

RULE
::  Ham   ->  [INT];
    Ham   ->  y: [1 | Merge (Map (* 2) y) (Merge (Map (* 3) y) (Map (* 5) y))];

::  Merge [INT]   [INT]     ->  [INT];
    Merge f       []        ->  f;
    Merge []      g         ->  g;
    Merge f:[a|b] g:[c|d]   ->  [a | Merge b g]     , IF < a c
                            ->  Merge f d           , IF = a c
                            ->  [c | Merge f d];
```

```
::  Map   (=> x y)   [x]    ->  [y];
    Map   f          []     ->  [];
    Map   f          [a|b]  ->  [f a | Map f b];

::  Take  INT        [x]    ->  [x];
    Take  0          list   ->  [];
    Take  n          [h|t]  ->  [h | Take (-- n) t];

::  Start  ->  [INT];
    Start  ->  Take 300 Ham;
```

The `Ham` function clearly has a cyclic structure. This solution to the Hamming problem runs in linear time, whereas in a language without the possibility to explicitly specify cyclic structures it is hard to even achieve a solution that runs in polynomial time.

1.7 Functions and constructors

In a language like Miranda a non-matching function will always result in a run-time error. In Concurrent Clean on the other hand non-matching functions are treated as constructors, as long as they do not result in type conflicts. The following example contains definitions that are used both as functions as well as as constructors.

```
MODULE Numerals;

TYPE
::  Num   ->  Zero
          ->  Succ Num
          ->  Pred Num;

RULE
::  Succ  Num       ->  Num;
    Succ  (Pred n)  ->  n;

::  Pred  Num       ->  Num;
    Pred  (Succ n)  ->  n;

::  Start           ->  Num;
    Start           ->  Succ (Succ Zero);
```

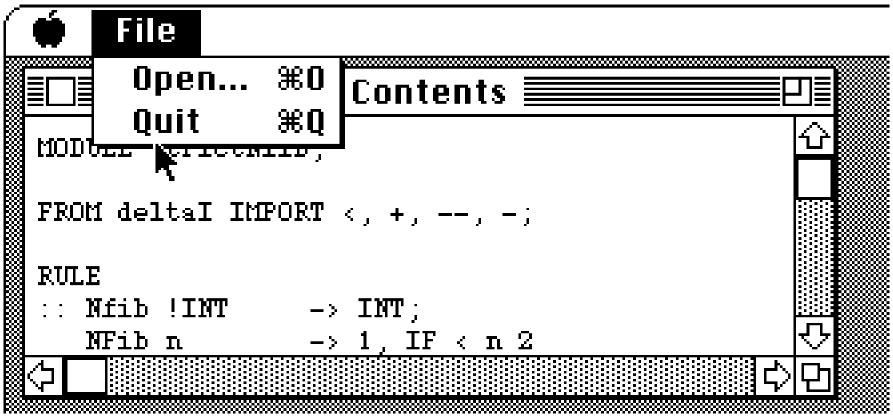
The right-hand-side of the `Start` rule will not match any rule. Still the program is correct, because the right-hand-side of the `Start` rule is indeed of the wanted type (`Num`). `Succ` and `Pred` are used as functions as well as as constructors. As constructor they are of type `Num` and as functions they yield type `Num`.

1.8 Input / Output

To be able to do window-based I/O a large I/O library has been implemented for Concurrent Clean. The implementation of the library is based on a kind of explicit environment passing scheme and guaranteed single threadedness by means of an extension of the type system with so-called unique types. Appendix C contains a complete overview of the functions and types that are predefined in the I/O library.

Interface objects (menus, windows etc.) are defined by means of parameterized constructors of well-chosen algebraic types. The behaviour of the interface objects is defined by its event handlers: program defined functions that will be called when a certain event occurs on that interface object. These event handlers work on a predefined, single threaded environment parameter: the `IOState`. This environment parameter takes care of the fact that the functional semantics are not violated.

This example shows some of the I/O features that are available in Concurrent Clean. It is a program that shows the contents of text files in a window. The program defines one menu, named ‘File’, containing two commands. The ‘Open...’ command presents a standard dialog for the selection of a file and then shows that file in the window. With the ‘Quit’ command one can quit the program. One window, called ‘Contents’, is defined by the program. The size of the window can be adjusted and the contents of the window can be scrolled in all directions to read the whole text. On the screen of an Apple Macintosh the program will present itself as follows:



For the sake of readability the comments in the program are not enclosed between comment brackets and are set in a different font. Constructors of predefined algebraic types are set bold, predefined functions and types are set in italics. Event handlers are set in bold and italics.

```
MODULE showfile;

IMPORT deltaI, deltaB, deltaFile, deltaWindow, deltaPicture;
IMPORT deltaEventIO, deltaIOSystem, deltaMenu, deltaFileSelect;
```

An interaction can be considered as a state transition system. The state of this transition system is formed by the program state State and the I/O state IO. The program state contains all global information needed during the interaction. In this case that is the state of the file system (modelled by the predefined environment FILES, which gives access to all files) and the contents of the file that is currently shown in the window. The I/O state is modelled by the predefined environment IOState, the environment parameter that makes it possible to do screen I/O.

```
TYPE
::  UNQ State    ->  (FILES,[STRING]);
::  UNQ IO       ->  IOState State;
```

The macro's that enhance comprehension and maintenance. Graphical interface objects have id's, which can be used to refer to the object during the interaction.

```
MACRO
    FileID      ->  1;            Corner    ->  (0,0);
        OpenID  ->  11;           Size      ->  (400,200);
        QuitID  ->  12;           MinSize   ->  (100,100);
                                  PicDom    ->  ((0,0),(500,300));
    WndwID      ->  1;            ScrBar    ->  ScrollBar (Thumb 0) (Scroll Space);
                                  Space     ->  12;
```

The Start rule is parameterized with the predefined WORLD environment, which contains all other environments. From the world the FILES and EVENTS environments are retrieved. The FILES environment is used to create the initial program state. The EVENTS environment, which models the event stream, is used together with the menu and window definitions to create the initial I/O state. StartIO starts the interaction. It terminates when one of the event handlers calls the predefined rule QuitIO (see the Quit function). When StartIO terminates the resulting EVENTS and FILES environments are put back into the world. The definition of the menu system is straightforward. The functions that will be called when the menu items are selected are Open and Quit. The (scrollable) window has two event handlers: Update is called when (parts of) the window need to be redrawn, Quit is called when the user requests to close the window.

```
RULE
::  Start  UNQ WORLD  ->  UNQ  WORLD;
    Start  world  ->  CloseEvents events' (CloseFiles files' world''),
        (files',cont):        state,
        (state,events'):      StartIO [menu,window] (files,[]) [] events,
        (files,world''):      OpenFiles world',
        (events,world'):      OpenEvents world,
        menu:      MenuSystem [PullDownMenu FileID "File"  Able [
                        MenuItem OpenID "Open..." (Key 'O') Able Open,
                        MenuItem QuitID "Quit" (Key 'Q')  Able Quit]],
        window:    WindowSystem [ScrollWindow    WndwID Corner "Contents"  ScrBar ScrBar
                                                 PicDom MinSize Size Update [GoAway Quit]];
```

Open is called when the Open... command is selected. It presents a predefined dialog to open a file (SelectInputFile), opens the file (SFOpen), reads it (ReadFile) and shows the contents in the window. First the size of the window's picture is adjusted (ChangeActivePictureDomain), after which drawing of the contents of the file can take place (DrawInActiveWindow). The font of the picture is set (SetFont), the previous content is erased (EraseRectangle) and the new content in shown (Display).

```
::  Open State IO -> (State,IO);
    Open (files,oldc) io
    ->  ((files1,oldc), io1), IF NOT read
    ->  (state', DrawInActiveWindow [SetFont font, EraseRectangle picdom | Display Space cont] io2),
        (state',io2):             ChangeActivePictureDomain picdom (files3,cont) io1,
        (len,cont,files3):        ReadFile file files2,
        (open,file,files2):       SFOpen name FReadText files1,
        (read,name,files1,io1):   SelectInputFile files io,
        picdom:                   ((0,0),(xsize, ysize)),
        xsize:                    + 20 (MaximumInList 0 ( FontStringWidths cont font)),
        ysize:                    + 20 (* Space len),
        (dummy,font):             SelectFont "Courier" [] 10;
```

Quit is called when the Quit command is selected or the window is requested to be closed. It simply applies the predefined function QuitIO to the current I/O state, which causes the interaction to terminate.

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

Update is called whenever (parts of) the window need to be redrawn. It calls the function Display to redraw the window's picture.

```
::  Update UpdateArea State -> (State, [DrawFunction]);
    Update area state:(files,cont) -> (state, Display Space cont);
```

ReadFile reads the lines of a file and closes it afterwards. It also returns the number of lines in the file. SFReadLine reads one line from the file and returns a new FILE on which reading can continue. SFEnd tests for end of file and SFClose closes the file.

```
::  ReadFile FILE FILES -> (INT,[STRING],FILES);
    ReadFile file files
    ->  (0, [], files1), IF SFEnd file
    ->  (++ len, [string | rest], files2),
        (dummy,files1):   SFClose file files,
        (len,rest,files2): ReadFile newf files,
        (string,newf):    SFReadLine file;
```

Display returns a list of draw functions that draw the list of strings in the picture of the window. MovePenTo moves the pen to the indicated position, DrawString draws a string at the current position of the pen.

```
::  Display INT [STRING] -> [DrawFunction];
    Display y []     -> [];
    Display y [s|r]  -> [MovePenTo (4,y), DrawString s | Display (+ y Space) r];
```

MaximumInList returns the maximum of a list of integers.

```
::  MaximumInList INT [INT] -> INT;
    MaximumInList max []            -> max;
    MaximumInList max [val | rest]  -> MaximumInList val rest , IF > val max
                                    -> MaximumInList max rest;
```

1.9 A parallel program

The sieve of Eratosthenes is a classical example in which a pipeline of parallel reducers can be created. Those reducers (`Sieve`'s) hold the prime numbers in ascending order, one in each `Sieve`. Each `Sieve` accepts a stream of integers as its input. Those integers are not divisible by any of the foregoing primes in the pipeline. If an incoming integer is not divisible by the local prime as well, it is sent to the next `Sieve`. A newly created `Sieve` accepts the first incoming integer as its own prime and outputs this prime and the channel of the next `Sieve` to a printing parallel reducer. After that it starts sieving. A parallel reducer called `Gen` sends a stream of integers greater than one to the first `Sieve`.

```
MODULE ParallelSieve;

IMPORT delta;

RULE
::  Start                        ->  [INT]
    Start                        ->  Sieve g,
                                     g: {P} Gen 2;

::  Sieve          [INT]         ->  [INT];
    Sieve          [pr | str]    ->  [pr  |  {P} Sieve ({I} Filter pr str)];

::  Filter  INT    [INT]         ->  [INT];
    Filter  pr     [n | str]     ->  Filter pr str      , IF = (% n pr) 0
                                 ->  [pr  |  {I} Filter pr str];

::  Gen     !INT                 ->  [INT];
    Gen     n                    ->  [n  |  {I} Gen (++ n)];
```

With only a few annotations (`{I}`, `{P}` and `!`) the desired parallel evaluation is achieved. The strictness annotation in the type rule of `Gen` is necessary to achieve efficiency. Leaving out this annotation would mean that a structure of applications of `++` to `2` would be sent to the other reducers by the `Gen` process, which is very inefficient. Leaving out all annotations results in a sequential program that will deliver the same result as the program of section 1.2.

2 Basic semantics of Concurrent Clean

Concurrent Clean is a lazy functional language based on Term Graph Rewriting Systems (Barendregt (1987a), Smetsers (1993)). This means that functions in a Concurrent Clean program work on graphs instead of terms (as is usual in other functional languages). In many cases the programmer does not need to be aware of the fact that he/she is manipulating graphs. But, for a good understanding of the underlying semantics of Concurrent Clean one needs some basic knowledge about graph rewriting.

2.1 Graph rewriting

A Concurrent Clean program basically consists of a number of graph rewriting rules which specify how a (program) graph has to be rewritten.

A *graph* is a set of nodes. Each node has a defining node-identifier (the *nodeid*). A *node* consists of a symbol and a (possibly empty) sequence of applied nodeid's (the *arguments* of the symbol). Applied nodeid's can be seen as references (arcs) to nodes in the graph, as such they have a *direction*: from the node in which the nodeid is applied to the node of which the nodeid is the defining identifier.

Each *rewrite rule* consists of a left-hand-side graph (the *pattern*) and a right-hand-side (rhs) consisting of a graph (the *contractum*) or just a single nodeid (a *redirection*). In Concurrent Clean rewrite rules are not *comparing*: the left-hand-side (lhs) graph of a rule is a tree, i.e. there exists exactly one path from the root to a node of this graph and each node identifier is applied only once.

A rewrite rule defines a (partial) *function*. The function name is the root symbol of the left-hand-side graph of the rule alternatives. All other symbols that appear in rewrite rules, are *constructors*.

The program graph is rewritten according to the rules. Initially this program graph is fixed: it consists of a single node containing the symbol `Start`, so there is no need to specify this graph in the program explicitly. The part of the graph that matches the pattern of a certain rewrite rule is called a *redex*. A *rewrite* of a redex can take place according to the right-hand-side of the corresponding rewrite rule. If the right-hand-side is a contractum then the rewrite consists of building this contractum and doing a redirection of the root of the redex to root of the right-hand-side. Otherwise only a redirection of the root of the redex to the single nodeid specified on the right-hand-side is performed. A *redirection* of a nodeid n1 to a nodeid n2 means that all applied occurrences of n1 are replaced by occurrences of n2.

A *reduction strategy* is a function that makes choices out of the available redexes. A *reducer* is a process that reduces redexes that are indicated by the strategy. The result of a reducer is reached as soon as the reduction strategy does not indicate redexes any more. A graph is in *normal form* if none of the patterns in the rules match any part of the graph. A graph is said to be in *root normal form* when the root of a graph is not the root of a redex and can never become the root of a redex.

The default reduction strategy used in Concurrent Clean is the *functional reduction strategy*. Reducing graphs according to this strategy resembles very much the way execution proceeds in lazy functional languages: if there are several rewrite rules for a particular function, the rules are tried in textual order; patterns are tested from left to right; evaluation of arguments is forced when it is tried to match an actual argument against a corresponding non-variable part of the pattern. A more formal definition of this strategy can be found in (Toyama (1991)).

Syntactical conventions

There are two kinds of identifiers, to wit variables and symbols. Variables begin with a lower-case character, symbols (functions, constructors, objects of a basic type, etc.) do not begin with a lower-case character. So all identifiers that begin with special characters like + or `$` are symbols. The alternative definitions of a function must be grouped together (separated by semi-colons), the keyword RULE indicates that a block of graph rewrite rules follows.

Basic syntax for rewrite rules

```
Rules           =    'RULE' Rule ';' {Rule ';'}
                ;
Rule            =    RuleAlt {RuleAlt}
                ;
RuleAlt         =    Graph '->' GuardedRhss {',' NodeDef}  ';'
                ;
GuardedRhss     =    GuardedRhs ['->' GuardedRhss]
                |    Expression
                ;
GuardedRhs      =    Expression [','] 'IF' Guards
                ;
Expression      =     NodeId
                |    [NodeId ':'] Node
                ;
Guards          =    Guards {'&&' Guards}
                |    Guards {'||' Guards}
                |    '(' Guards ')'
                |    Expression
                ;
Graph           =    [NodeId ':' ] Node {',' NodeDef}
                ;
NodeDef         =     Nodeid ':'  Node
                ;
Node            =    Symbol {Arg}
                ;
Arg             =     Nodeid
                |    [Nodeid ':'] Symbol
                |    [Nodeid ':'] '(' Node ')'
                ;
Nodeid          =    Variable
                ;
```

Instead of the readable shorthand notation that was used in the examples of chapter 1, the syntax also allows to use a notation in which the graph structure has been made more explicit as can be seen in the example in one of the following sections.

Static restrictions

An *explicit* definition of a node identifier (nodeid) is an occurrence before a colon symbol in a node definition. An *implicit* definition of a nodeid is any occurrence on the left-hand-side of a rule that is not an explicit definition. An occurrence of a nodeid is a defining occurrence if it is an occurrence of an explicit or implicit definition. All other occurrences of nodeid's are applied occurrences. The scope of the node identifier is the alternative (consisting of one or more guarded expressions) of the rule in which the node identifier occurs. In rewrite rules each nodeid must have one unique defining occurrence within its scope.

Arities and Currying

In Concurrent Clean all symbols are *defined* with a fixed arity. However, a symbol may be *applied* on an arbitrary number of arguments. If a function is applied with too few arguments the expression is in root normal form. When all arguments of a function are present it can be reduced.

When a function F is defined with arity n:

$F\ a_1\ a_2\ \ldots\ a_n \rightarrow a_r$

then

```
(...((F a1) a2)...) an
```

reduces to

```
F a1 a2 ... an
```

which reduces to

```
ar
```

The example in section 1.3 (the `Map` function) shows the use of curried functions. Another example is the `foldr` function:

```
::   Foldr (=> t1 (=> t2 t2)) t2 [t1] -> t2;
     Foldr op r []       -> r;
     Foldr op r [a|x]    -> op a (Foldr op r x);
```

The first argument of the higher order function `Foldr` (`op`) is a function that is applied twice on the left-hand-side. Once with two arguments, in which case it can be reduced, and once with zero arguments, in which case it cannot be reduced: it is in root normal form.

A small example

Consider the following rewrite rules:

```
Add Zero z          →    z;                                    (1)
Add (Succ a) z      →    Succ (Add a z);                       (2)

Start               →    Add (Succ o) o,
                         o: Zero;                              (3)
```

These rules are semantically equivalent to the following set of Concurrent Clean rules in which the graph structure has been made explicit by adding nodeid's. Using the set of rules with explicit nodeid's it will be easier to understand what the meaning is of the rules in the graph rewriting world.

```
x: Add y z,
y: Zero             →    z;                                    (1)
x: Add y z,
y: Succ a           →    m: Succ n,
                         n: Add a z;                           (2)

x: Start            →    m: Add n o,
                         n: Succ o,
                         o: Zero;                              (3)
```

The fixed initial program graph that is in memory when a program starts is the following:

The initial graph in linear notation:

```
@DataRoot : Graph @StartNode,
@StartNode: Start
```

The initial graph in pictorial notation:

```
@DataRoot:Graph
      ↓
@StartNode:Start
```

To distinguish the nodeid’s appearing in the rewrite rules from the nodeid’s appearing in the graph the latter always begin with a ‘`@`’.

The initial graph is rewritten until it is in normal form. Therefore a Concurrent Clean program must at least contain a "start rule" that matches this initial graph via a pattern. The right-hand-side of the start rule specifies the actual computation.

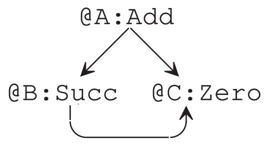
The patterns in rewrite rules contain formal nodeid's. During the matching these formal nodeid's are mapped to the actual nodeid's of the graph. After that the following semantic actions are performed:

The start node is the only redex matching rule (3). The contractum can now be constructed:

The contractum in linear notation:

```
@A: Add  @B @C,
@B: Succ @C,
@C: Zero
```

The contractum in pictorial notation:

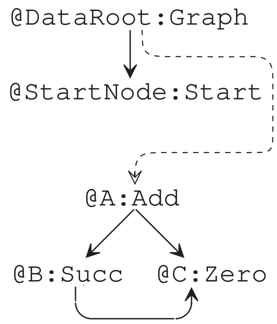


All applied occurrences of `@StartNode` will be replaced by occurrences of `@A`. The graph after rewriting is then:

The graph after rewriting:

```
@DataRoot : Graph @A,
@StartNode: Start,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero
```

Pictorial notation (redirected nodeid is dashed):

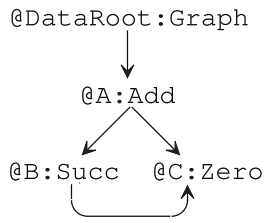


This completes one rewrite. All nodes that are not accessible from `@DataRoot` are garbage and not considered any more in the next rewrite steps. In an implementation once in a while garbage collection is performed in order to reclaim the memory space occupied by these garbage nodes. In this example the start node is not accessible from the dataroot node after the rewrite step and can be left out.

The graph after garbage collection:

```
@DataRoot : Graph @A,
@A: Add  @B @C,
@B: Succ @C,
@C: Zero
```

Pictorial notation :

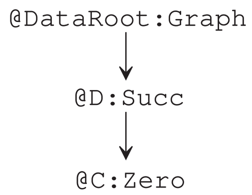


The graph accessible from `@DataRoot` still contains a redex. It matches rule 2 yielding the expected normal form:

The final graph:

```
@DataRoot : Graph @D,
@D: Succ @C,
@C: Zero
```

Pictorial notation :



The fact that graphs are being used in Concurrent Clean gives the programmer the ability to explicitly share terms or to create cyclic structures. In this way time and space efficiency can be obtained.

2.2 Guarded expressions

In Concurrent Clean conditional expressions are specified by means of guards. Guards should be considered as an extension of the notion of patterns: an alternative only matches when its pattern matches and its guard evaluates to True. It is important to note that pattern matching always takes place before the guards are evaluated. A group of alternatives with the same left-hand-side (pattern) is considered as one alternative consisting of one or more guarded expressions. The left-hand-side of the guarded expressions need only be specified once before the first guarded expression. Guarded expressions are separated by arrow symbols ('->'). When the last guarded expression has a guard that always evaluates to True that guard can be omitted (the 'otherwise' or 'default' case).

Guards are separated from the rest of a guarded expression by an optional comma and the keyword `IF`. A guard is a boolean expression in which the infix operators `||` (conditional or) and `&&` (conditional and) can be used. These operators have no priority and are not allowed outside guards. Parentheses can be used in guards to specify the priorities. The operators `||` and `&&` are conditional, which means that when their result can be calculated by evaluating their first operand only, the second operand will not be evaluated. For example, when the first operand of `||` evaluates to True, the second operand will not be evaluated: the result will be True anyway.

Note that semantically a guarded expression should be considered as an individual rule alternative. For instance, local annotations (see chapter 6) will be considered only if the corresponding guard evaluates to True. The same holds for unique objects (see section 3.6): a unique object can be used in several different guarded expressions without losing its uniqueness property.

Example of the use of guards: the Merge function (merge two sorted lists of integers, remove doubles):

```
::  Merge [INT]    [INT]      ->   [INT];
    Merge []       g          ->   g;
    Merge f        []         ->   f;
    Merge f:[a|b]  g:[c|d]    ->   [a | Merge b g], IF < a c
                              ->   Merge f d      , IF = a c
                              ->   [c | Merge f d];
```

2.3 Lists and tuples

To provide some programming convenience Concurrent Clean provides two predefined (generic) data structures, namely lists and tuples. Below the syntax is given. The list notation in Concurrent Clean is almost similar to notations in other functional languages: `[1,2,3,4]` denotes a list with the four integers 1, 2, 3 and 4, `[]` is an empty list, `[f | r]` denotes a list `r` prefixed with an element `f` and `[e1,...,en | r]` denotes a list `r` prefixed with elements `e1...en`. Tuples are denoted identically to many other functional languages: the elements of a tuple are listed between parentheses and separated by comma's. An example of a tuple is `(1,'?',FALSE)`.

```
Graph         =    [NodeId ':'] Node {',' NodeDef}
              ;
NodeDef       =    NodeId    ':' Node
              |    NodeIds   ':' ListNode
              |    NodeIds   ':' ApplyNode
              |    NodeIds   ':' SimpleNode
              |    NodeIds   ':' NodeId
              ;
NodeIds       =    '(' NodeId ',' NodeId {',' NodeId } ')'
              ;
Node          =    TupleNode
              |    ListNode
              |    ApplyNode
              |    SimpleNode
              ;
```

```
TupleNode      =    '(' ListOrTupleArg ',' ListOrTupleArg
                         {',' ListOrTupleArg} ')'
               ;
ListNode       =    '[]'
               |    '[' ListOrTupleArg {',' ListOrTupleArg }
                         ['|' ListOrTupleArg] ']'
               ;
ApplyNode      =    NodeId Arg {Arg}
               ;
SimpleNode     =    Symbol {Arg}
               ;
Arg            =    NodeId
               |    [NodeId':'] ArgNode
               ;
ArgNode        =    Symbol
               |    '(' Node ')'
               |    TupleNode
               |    ListNode
               ;
ListOrTupleArg=     NodeId
               |    [ NodeId':'] Node
               ;
```

From the syntax specification it follows that it is allowed to specify a definition of a number of nodeid's simultaneously by placing a tuple of nodeid's before a colon symbol in a node definition. The meaning of such a construction is that all these nodeid's are bound to parts of the graph on the right-hand-side of the colon symbol at run-time. Note that this only makes sense if this graph denotes a function that delivers a tuple. This will be verified by the type checker (see chapter 3).

The following program contains an example of the use of tuples before a colon symbol in a node definition.

```
RULE

<<   FindElem traverses a list of lists of integers to search
     for the element el. The list containing the element and
     its index in this list  is returned.
>>

::   FindElem [[INT]] INT    -> ([INT],INT);
     FindElem [l | ls] el    -> (l,index)     , IF found
                             -> FindElem ls el,
                                (found,index): Index l el 1;

::   Index [INT] INT INT     -> (BOOL,INT);
     Index [] el n           -> (FALSE, n);
     Index [f | r] el n      -> (TRUE, n)         , IF = el f
                             -> Index r el (++ n);

::   Start                   -> ([INT],INT);
     Start                   -> FindElem [l1,l2,l3] 6,
                                l1: [2,3,5,7,11,13,17,19],
                                l2: [-9,-6,-3,0,3,6,9],
                                l3: [1,4,9,16,25,36];
```

2.4 Macro's

A disadvantage of using a function definition to assign a name to a constant is that these functions cannot be used for this purpose in the patterns on the left-hand-side. Though it is not forbidden in Concurrent Clean to use function symbols in patterns there is not much point in specifying a pattern of which an instance is a redex. Because the functional strategy will always reduce that redex before the pattern matching takes place the pattern will never match the graph. This is the main reason why Concurrent Clean has been extended with so-called *macro's*: special rewrite rules that are rewritten (*expanded*) at compile-time.

Apart from the above-mentioned advantage of macro's it should be mentioned that the use of macro's instead of non-recursive functions can speed up a Concurrent Clean program because less function calls need to be done. A disadvantage is that more code will be generated when (parameterized) macro's are used instead of non-recursive functions.

Besides `TYPE` blocks and `RULE` blocks also `MACRO` blocks may occur in a Clean module. A `MACRO` block consists of a number of macro definitions of the form `Left-hand-side-graph -> Right-hand-side-graph;`. At compile time the right-hand-side of the macro definition will be substituted for every occurrence of the left-hand-side. Macro's may have parameters but it is not allowed to specify a pattern on the left-hand-side of a macro definition. Consequently, a macro rule always consists of a single alternative. It is clear that, at compile-time, the rewriting of the macros should always terminate. This is guaranteed by prohibiting macros that are mutually dependent or recursive.

Example of the use of macro's in Concurrent Clean.

```
MODULE macro;

TYPE
...

MACRO
     Size               -> 6;
     UnsortedList       -> [3,1,5,2,4];
     Singleton x        -> [x];
     Pair a b           -> (a,b);
     +SC string char    -> +S string (CTOS char);
     *R4 r1 r2 r3 r4    -> *R r1 (*R r2 (*R r3 r4)));

RULE
...
```

Macro definitions are similar to synonym type definitions (see section 3.4). The expansion of synonym types also takes place at compile-time and recursively defined synonym types are forbidden too. The major difference between macro's and synonym types is caused by the way expansions are performed. In the first case graph reduction is used whereas the in second case string reduction is used. Example:

```
MACRO
     Double x  -> + x x;

TYPE
::   Twice x   -> (x, x);
```

The result of expanding the macro `Double` in the application '`Double (Fac 100)`' is '`+ x:(Fac 100) x`' whereas the result of expanding `Twice` in '`Twice [INT]`' is '`([INT],[INT])`'.

The expansion of the macro's takes place before type checking. Type specifications of macro rules are not allowed.

3 The type system

Concurrent Clean is a typed language (Bakel (1991)). With type information a compiler can generate much more efficient code for rewrite rules, in particular when basic types are used in combination with strictness annotations (see chapter 6). Furthermore type information makes separate compilation possible which is of great practical importance.

Since Concurrent Clean is based on functional Graph Rewriting Systems, it was possible to adopt the polymorphic type system that is also used in Miranda (the Milner/Mycroft (Milner (1978), Mycroft (1984)) scheme). Like in Miranda, types do not have to be specified explicitly by the programmer. However, the programmer is strongly advised to always specify the type of the objects. For programmers in a hurry, types can be inferred by the Concurrent Clean compiler if type information is left out. For programmers who do not worry (and for hackers), type inferring/checking can be switched off. For future versions of Concurrent Clean we plan a type system based on intersection types (Bakel (1993)) that will be more liberal than the Milner/Mycroft scheme.

3.1 Predefined basic types

For practical reasons it is convenient that rules for performing operations on objects of primitive types (integers, reals etc.) are predefined such that they can be implemented efficiently, preferably by using the representation and corresponding operations available on the computer.

In Concurrent Clean the following basic types are available (see also appendix B):

Basic type:	Examples of denotations:	Examples of predefined functions:
INT	1, 2, -56438	+, -, =, >=, AND%
REAL	1.5, 0.314E10	+R, -R, =R, >=R, SIN, EXP
CHAR	'a', 'b', 'c', '\n', '\007'	+C, -C, =C, >=C
BOOL	TRUE, FALSE	NOT, AND, XOR, =B
STRING	"This is a string \n"	+S, -S, =S, >=S, SLICE

3.2 Predefined type constructors

Lists and tuples are predefined (generic) data structures of Concurrent Clean. For the denotation of list and tuple types special type constructors have been predefined. For lists square brackets (‘[‘ and ‘]’) and for tuples round brackets (‘(‘ and ‘)’) are used. For example: `[INT]` denotes a list of integers and `(CHAR,BOOL)` denotes a tuple with a character and a boolean.

To be able to specify the type of higher order functions the predefined type constructor ‘=>’ exists. This is a prefix, dyadic type constructor that denotes the type of a (curried) function with one argument. By means of currying, functions with more than one argument can be typed. For example: `=> INT (=> INT INT)` denotes the type of a dyadic function working on integers when this function is passed as a parameter to a higher order function.

Of course the constructors described above can be used nested. As an example the type specification is given of a function `F` that maps a function over a list of pairs, returning a list of result values:

```
::   F (=> (x,y) z) [(x,y)] -> [z];
```

3.3 Type definitions

The objects that are manipulated in Concurrent Clean are graphs. Type definitions are therefore rules that specify certain restrictions imposed on the graphs on which rewrite rules are defined. Syntactically, types are defined by graph rewrite rules. Semantically there is of course a big difference between type rules and rewrite rules. In fact, in Concurrent Clean types specify restrictions on terms, rather than on graphs.

Therefore, all graphs in type definitions have to be directed acyclic graphs (dag's) without explicit sharing of nodes.

Syntax for defining types:

```
CleanProg       =    {Block}
                ;
Block           =    TypeBlock
                |    MacroBlock
                |    RuleBlock
                ;
TypeBlock       =    'TYPE'    TypeRule  {TypeRule}
                ;
TypeRule        =    '::' TypeAlt {'|'  TypeGraph} ';'
                |    '::' TypeAlt {'->' TypeGraph} ';'
                ;
TypeAlt         =    TypeGraph '->' TypeGraph
                ;
TypeGraph       =    Graph
                ;
```

Types are defined in a so-called `TYPE`-block, just like rewrite rules are defined in a `RULE`-block and macro's are defined in a `MACRO`-block. `TYPE`-blocks, `MACRO`-blocks and `RULE`-blocks may be specified in any order. Each type rule must be preceded by a `::`.

The root symbol of the type rule is called the *type symbol*. The nodeid's that occur in type rules are now *type variables*. Each time the same variable is used in a particular type rule, it stands for the same type. So with the aid of a new type definition containing type variables a set of types can be made rather than just a single type.

In Concurrent Clean we distinguish type definitions for introducing new types and type specifications for functions.

3.4 New type definitions

For a type definition that introduces a new type, the left-hand-side graph consists of a single node with no sharing of arguments. The symbol of this node is the *type symbol*. It defines the name of the type.

As said before, the nodeid's that occur in type rules are now *type variables*. Each type variable must have exactly one implicit definition, i.e. there must be one applied occurrence of this variable on the left-hand-side of the type rule.

There are three kinds of type definitions that introduce new types: *synonym* type definitions, *algebraic* type definitions and *abstract* type definitions.

Synonym types

Synonym types are specified by means of a type rule having exactly one alternative of which the right-hand-side is a type instance. Mutual dependencies of synonym type definitions are prohibited.

Synonym types permit the user to introduce a new name for an already existing type.

Example of a type synonym definition in Concurrent Clean:

```
TYPE
::   Stack x        ->   [x];
```

A more complicated example introduces a generic type that can be used for specifying the type of a dyadic operator over arbitrary types:

```
TYPE
::   DiOpType x     ->    => x (=> x x);

RULE
::   Map2 (DiOpType x) [x] [x]    -> [x];
     Map2 op [] []                -> [];
     Map2 op [f1 | r1] [f2 | r2]  -> [op f1 f2 | Map2 op r1 r2];

::   Start          -> INT;
     Start          -> Map2 * [2,3,4,5] [7,8,9,10];
```

Algebraic types

Algebraic types are specified by means of a type rule whereof each alternative has a right-hand-side with a unique root symbol: the *constructor*. All the arguments of the constructor are type instances (see below).

A *type instance* is either a type variable or an acyclic graph that has a root symbol that is a type symbol of which all the arguments are type instances.

If a *constructor* appears in the right-hand-side of a algebraic type rule, the constructor is said to be of that specific type.

Example of algebraic type definitions in Concurrent Clean. The types `Nat`, `Bool`, `List` and `Combi` are defined. The constructor `Zero` is said to be of type `Nat`. The constructor `Succ` of type `Nat -> Nat` etcetera.

```
TYPE
::   Nat       ->   Zero
               ->   Succ Nat;

::   Bool      ->   True | False;

::   List x    ->   Cons x (List x)
               ->   Nil;

::   Combi x   ->   Pair1 Nat x
               ->   Pair2 x (List x)
               ->   Pair3 Bool Bool;
```

Alternatives of an algebraic type definition are separated by an arrow symbol (‘`->`’) or by a vertical bar (‘`|`’).

Abstract types

Abstract types are specified by means of a left-hand-side graph in a definition module only.

To distinguish an abstract type definition from an ordinary type definition a special kind of type block is provided, called an `ABSTYPE`-block.

Example of abstract type definitions in Concurrent Clean, assuming the algebraic type definitions of the examples above:

```
ABSTYPE
::   NatList;

::   Stack x;
```

In Concurrent Clean, abstract type definitions are only allowed in definition modules (see chapter 4).

An elaborate example of an abstract data type:

```
DEFINITION MODULE Stack;

ABSTYPE
::   Stack x;

RULE
::   Empty                        ->   Stack x;
::   IsEmpty   (Stack x)          ->   BOOL;
::   Push      x (Stack x)        ->   Stack x;
::   Pop       (Stack x)          ->   Stack x;
::   Top       (Stack x)          ->   x;
```

The concrete definition (*signature*) of the abstract type has to be defined in the implementation module as either a synonym type or an algebraic type. This concrete definition is invisible for the outside world. Therefore, for such an abstract data type functions have to be provided that create and manipulate objects of the abstract type.

The signature of the abstract type `Stack`:

```
IMPLEMENTATION MODULE Stack;

IMPORT delta;

TYPE
::   Stack x   -> [x];

RULE
::   Empty                                  ->   Stack x;
     Empty                                  ->   [];

::   IsEmpty   (Stack x)                    ->   BOOL;
     IsEmpty   []                           ->   TRUE;
     IsEmpty   s                            ->   FALSE;

::   Push      x (Stack x)                  ->   Stack x;
     Push      e s                          ->   [e | s];

::   Pop       (Stack x)                    ->   Stack x;
     Pop       [e | s]                      ->   s;

::   Top       (Stack x)                    ->   x;
     Top       [e | s]                      ->   e;
```

Note that in fact all predefined types can be seen as special cases of abstract types. In chapter 4 more information on definition and implementation modules can be found.

3.5 Type specifications for functions

Each rewrite rule can be typed explicitly by the programmer. This type specification must immediately precede the corresponding rewrite rule.

Syntax for Concurrent Clean rewrite rules with types:

```
RuleBlock      =    'RULE' [TypeRule ';'] Rule ';'
                           {[TypeRule ';'] Rule ';'}
```

A type specification for a rewrite rule is a type rule with the following additional restrictions:

- the type symbol of the type rule is the same as the function symbol of the corresponding rewrite rule. Both symbols must have the same number of arguments;

- the right-hand-side as well as the arguments on the left-hand-side are type instances;
- the type rule consists of one rule alternative;
- type variables on the right-hand-side that have no defining occurrence are by definition implicitly defined. This means that new type variables may be introduced on the right-hand-side of a type specification which is not allowed in type definitions (see section 3.3)
- the type specification is correct (see below).

In Concurrent Clean a type specification is defined to be *correct* (with respect to the Milner/Mycroft type scheme):

- if for each argument of each symbol the *demanded* type (i.e. the type that is indicated in the corresponding type specification) is *unifiable* (i.e. there exists a type that is a type instance of both types) with the *offered* type (i.e. the actual type of the argument);
- if for each right-hand-side the offered type is unifiable with the right-hand-side of the corresponding type specification.

The correctness of the type specifications and the rewrite rules is checked at compile time by the type checker. Conflicts are reported by the compiler and result in an error. The compiler has as special feature the option to switch type checking off (see section 3.7) which also disposes of the correctness demand of types.

Example of implicitly defined nodeid's on the right-hand-side of a type definition for a rewrite rule:

```
::  F    ->   [t];
    F    ->   [];
```

`F` has as result type any list.

To illustrate typing in Concurrent Clean an example is given that reverses a list twice:

```
MODULE twice_reverse;

IMPORT delta;

RULE

::  Reverse [t]           -> [t];
    Reverse [x | r]       -> Append (Reverse r) x;
    Reverse []            -> [];

::  Append [t] t          -> [t];
    Append [x | r] y      -> [x | Append r y];
    Append [] y           -> [y];

::  Twice (=> t t) t      -> t;
    Twice f x             -> f (f x);

::  Start                 -> [INT];
    Start                 -> Twice Reverse (FromTo 1 100);
```

Note that the types in the example are similar but not exactly the same as they would have been in Miranda. Since functions are not implicitly curried in Concurrent Clean, `Append` is a function with two arguments and not a function with one argument that yields a function that takes the following argument as would have been the type of a similar definition in Miranda. So, the type definition of `Append` must be read as: `Append` is a function with two arguments of type `[t]` and `t` and the result of `Append` is of type `[t]`. For an explanation of the type of `Twice` see the section on typing curried functions.

Typing partial functions

Some type errors cannot be detected at compile-time. If a function is a partial function there is a chance that such a function is called with an actual argument of the correct type (domain) for which no image exists, due to a failing pattern match. An error is generated at run-time if this leads to a wrong type.

Consider the following example:

```
RULE
::   F INT     ->   INT;
     F 0       ->   0;

::   Start     ->   INT;
     Start     ->   F 1;
```

Although the Concurrent Clean program is correctly typed, the function `F` applied in the start-rule cannot be matched and therefore `F 1` will not yield the required type: `INT`. At run-time, an error is generated.

However, a failing match does not always have to be wrong. A match may fail as long as no type conflicts are caused by this failing.

Consider the following example:

```
TYPE
::   Num  ->   Zero
          ->   Succ Num
          ->   Pred Num;

RULE
::   Succ Num        ->   Num;
     Succ (Pred n)   ->   n;

::   Pred Num        ->   Num;
     Pred (Succ n)   ->   n;

::   Start           ->   Num;
     Start           ->   Succ (Succ Zero);
```

The graph `Succ (Succ Zero)` in the start rule will not match any rule. Still it is correct because the graph is indeed of the wanted type (`Num`). Notice that `Succ` and `Pred` are used both as functions as well as constructors. As constructors they may appear in the right-hand-side of type definitions and are of type `Num`. As functions they also yield type `Num`.

In general graph rewriting non-matching functions are treated as ordinary constructors. In Concurrent Clean they are acceptable as long as they do not result in type conflicts. The precise conditions under which functions are allowed in patterns can be found in (Bakel (1991)). In a language like Miranda a non-matching function will always result in a run-time error.

Typing curried functions

In Concurrent Clean all symbols are defined with fixed arity. However, in a symbol application it is allowed to use it with an arbitrary number of arguments (see also section 2.1). With the aid of the predefined internal function `_AP` a curried function applied on the needed amount of arguments is transformed into an equivalent uncurried function application. This facility implies the need for a special predefined type, the *curried function type* denoted by =>. This predefined type can be compared with the function type as predefined in Miranda, used in prefix notation. So, the type `=> a b` stands for a curried function (of arity one) that, when applied to an argument of type `a` will produce a result of type `b` (pronounce `=> a b` as: curried function from `a` to `b`).

The type axiom's of the Concurrent Clean type system include the equivalence of

```
::   F -> => t1 (=> t2 (=> ... (=> tn tr) ... ))
```

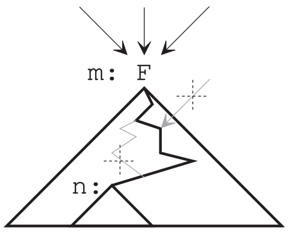
with

```
::   F t1 t2 ... tn  -> tr;
```

if `F` is defined with arity `n`.

3.6 Unique types

A node *n* of a graph *G* is *unique* with respect to a node *m* of *G* if *n* is only reachable from the root of *G* via *m* and there exists at most one path from *m* to *n*.



An important property of a unique node is the fact that it has a reference count of one. A shared node which has a larger reference count can never be unique with respect to any node. A reference count of one is however not sufficient for uniqueness with respect to any other node.

Unique nodes are very important, because they make it possible to implement side-effects efficiently, without loosing the functional properties of the program. For instance, assume that a node is passed as argument of a certain function application in such a way that the node is unique with respect to that function application: if such a node is accessed via a variable in a pattern of the corresponding rewrite rule and that variable is not used on the right-hand-side of that rule, it can be considered as garbage and reused for building the contractum. Among other things, this technique can be used for the implementation of functions to update an array destructively. Also writing to a file or window can be done instantaneously, without any overhead at run time.

It would be nice if at compile time the uniqueness of arguments and results of functions could be determined. Unfortunately, this is undecidable. In Concurrent Clean a decidable approximation has been incorporated using **unique types** (Smetsers et al (1993)). Unique types, defined on graphs, have some similarities with linear types, defined on lambda terms (Girard (1987), Wadler (1990)).

UNQ type specifications and functions

The type of a graph in a rewrite rule can have the unique attribute, i.e. the graph can be of type `UNQ T`. *If a graph in a left-hand side or in the right-hand side of a rewrite rule is of type* `UNQ T`*, it is guaranteed that at run-time the corresponding root of the graph is unique with respect to the root of respectively the function application or contractum.* When this cannot lead to confusion we will use the phrase "a unique graph" in stead of "a graph which is unique with respect to the function application cq the contractum".

The `UNQ` type attribute can be added by the programmer to any type to express the restricted use of a function. To verify the correctness of the use of `UNQ` attributes the type system has been extended. This means that all applications on the right-hand-side of a function are examined to check that when a parameter or a result of an `UNQ` type is demanded, a unique graph of the demanded type is offered. Here, demanded means that either the corresponding formal parameter of the applied function has an `UNQ` attribute or the result type of the defined function itself is attributed with `UNQ`. To get a well-typed program, the following four general rules have to be obeyed:

1. Demanded `UNQ`'s have to be obeyed: when a demanded type `T` and an offered type `T'` are unifiable (in the standard sense, ignoring the `UNQ` attributes) then for each position where in `T` the `UNQ` attributes occurs, there has to be a corresponding position in `T'` with the `UNQ` attribute.

   Not well-typed example:

```
::  F    UNQ INT   -> INT;
    F    n         -> F (G n);

::  G    INT       -> INT;
    ...
```

   In the second alternative, `F` is recursively called with a graph of type `INT`, while in the type specification of `F` an argument of type `UNQ INT` (which is more restrictive) is demanded. This will yield a type error. Notice that the application `G n` is correct. `G` demands an argument of type `INT` while `G` is applied with a graph of type `UNQ INT`. Since `INT` is less restrictive than `UNQ INT`, this is ok.

2. An applied occurrence of a node-id bound to a graph can have the `UNQ` type attribute only if there exists at most one path from the root of the right-hand-side of a rule alternative to that applied occurrence.

   Not well-typed example: if a function `F` is specified with the following type:

```
::  F UNQ INT      ->   ([UNQ INT], [UNQ INT]);
    F x            ->   (y, y),
                        y: [x];
```

   The type specification will be rejected since two paths exist leading from the root of the contractum to the same node `x`. The unique argument of the function `F` will not be unique anymore in the resulting pair.

   Each function alternative is separately checked for consistency. Example:

```
::  F    UNQ INT   INT  -> UNQ INT;
    F    n         m    -> n, IF = m 0
                        -> F (G n) (-- m);

::  G    UNQ INT        -> UNQ INT;
    ....
```

   Although in the definition of `F` the unique node-id `n` is used twice this does not lead to a conflict (because only one of the guarded expressions will be chosen at run-time).

   Not well-typed example:

```
::  F    UNQ INT   INT  -> UNQ INT;
    F    n         m    -> n, IF = m 0
                        -> F n (G n);
```

   Here we clearly have an erroneous situation: in the recursive call of `F` there are two paths from the root of the contractum to the node-id `n`. So, in this recursive call the first argument `n` does not satisfy the specified type of `F`.

It is not always wrong to use an `UNQ`-typed graph more than once in one and the same function alternative. A unique node-id may be shared between the guard and the guarded expression belonging to it. A guard is formally not a part of the right-hand-side of a function definition. A guard is evaluated in advance to determine which of the function alternatives have to be chosen. A guard will yield a boolean result that cannot contain references to `UNQ` nodes. To guarantee that no side-effects occur when evaluating a guard, a graph that is being used in both guard and guarded expression will loose its unique type attribute in the guard. This enables the non-destructive (observing) inspection of such a graph in a guard.

Example:

```
::  F    UNQ INT   -> UNQ INT;
    F    n         -> n, IF = n 0 || = n 1
                   -> F (G n);
```

Although the node-id n is shared between the guard and both guarded expressions this will not lead to an erroneous situation. After the evaluation of the guard still only one non-garbage reference to n exists. The type checker will assign the type INT (and not UNQ INT) to both uses of n in the guard.

3. Offered function types with UNQ arguments have to be obeyed: when a demanded type => T T' and an offered type => S S' are unifiable (in the standard sense, ignoring the UNQ attributes) then for each position where in T and T' the UNQ attribute occurs for an argument of a function type, there has to be a corresponding position in S and S' with the UNQ attribute.

4. All occurrences of the same type variable must uniformly be specified with the same type attribute.

Example of rules 3 and 4:

```
::  UMap (=> UNQ x UNQ y) [UNQ x]     ->   [UNQ y];
    UMap uf               [ ]         ->   [ ];
    UMap uf               [ux | uxs]  ->   [uf ux | UMap uf uxs];

::  WriteAll CHAR [UFILE]   -> [UFILE];
    WriteAll c    ufiles    -> UMap (FWriteC 'c' ufiles);
```

This variant of Map is defined on functions that take a graph of type UNQ x and yield a graph of type UNQ y. UMap applies such a function to a unique list (see below) with elements of type UNQ x yielding a unique list with elements of type UNQ y.
This function UMap is used in the function WriteAll to map a function FWriteC that writes a character into a unique file on a list of unique files.
It has to be guaranteed that the function FWriteC is always applied to a unique file. From the type of the standard function Map one cannot deduce whether this condition is fulfilled. Therefore in this example the use of Map instead of UMap would have been rejected. Notice that there is no distinction between the definition of the function Map and UMap, there is only a difference in the specified type.

In the current version of Concurrent Clean functions cannot be defined polymorphic in their UNQ attributes. Such a feature would enable the definition of a generic Map function that can be used both as the standard Map function and as the UMap function in the example above. In the next version of Clean this will be changed.

Defining types with UNQ attributes.

A programmer can add unique type attributes to any type. In particular, type attributes can be used in the definition of new types (algebraic, abstract and synonym types). In this section it is explained what the meaning of such a definition is. As explained below, an important aspect of a unique graph is that it cannot be a part of a graph that is not unique.

Predefined type constructors

List, tuples and function types, as well as their sub-types, can have the unique attribute. However, when a graph with type T contains unique parts (which means that the type T has unique subtypes) the graph itself must be unique (this is called the *UNQ propagation rule*). Otherwise, there can be multiple paths leading to this graph which means that its components would not be unique anymore.

The propagation rule holds for all types, also for function types. For non-function types UNQ attributes propagate from "inside-out", for function types they propagate from “left-to-right”.

Example of the UNQ propagation rule for tuples (analogue for lists):

"inside-out" propagation: Consider the type (INT, UNQ CHAR). An instance of such a type is a pair containing an INT and a unique CHAR. Due to the propagation rule, the pair itself has to be unique as well, i.e. the full type specification is: UNQ (INT, UNQ CHAR).

"left-to-right" propagation: Example of `UNQ` propagation for function types. Assume that `F` is defined as:

```
:: F UNQ INT  INT -> INT;
```

The type of the curried function `F` is: `=> UNQ INT (=> INT INT)`. Now consider the curried application `F 3`. This application has taken a unique integer value. When there are multiple paths leading to `F 3` there are also multiple paths leading to `3`. So, the full type specification for `F` is: `=> UNQ INT UNQ(=> INT INT)`.

In Concurrent Clean it is not necessary to give the full type specification; the propagation of the `UNQ` attributes is performed automatically by the compiler.

Synonym types

Synonym types can be used in the usual way.

Example:

```
TYPE
::   UTUPLE UNQ x -> (UNQ x, UNQ x);
```

Note that this is not the full type specification. Applying `UNQ` propagation to achieve the full type specification gives `UNQ UTUPLE UNQ x -> UNQ (UNQ x, UNQ x)`.

Algebraic types

The following rules guarantee that when unique sub-structures are being used, a consistent (recursive) algebraic type is being defined.

1. In algebraic types all occurrences in both left- and right-hand-side of a certain type variable must have the same type attribute, just like in function types.
2. All occurrences of a recursive algebraic type must be attributed uniformly.
3. When the root of a right-hand-side of a type alternative is attributed with `UNQ`, all other type alternatives will get the `UNQ` attribute on their root as well. Moreover, the algebraic type itself (and hence all occurrences of that type) will get the `UNQ` attribute.

Example: The following definition leads to lists that have a unique spine while the elements of such a list need not to be unique.

```
::   List x    ->   Cons x UNQ (List x)
               ->   Nil;
```

Due to the `UNQ` propagation rule the uniqueness of the spine will propagate over the `Cons`. As a consequence, `Nil` will get the `UNQ` attribute (being the root of the other type alternative) as well as all occurrences of `List x` (being the defined algebraic type). So, the above definition is equivalent with:

```
::   UNQ List x     ->   UNQ Cons x UNQ (List x)
                    ->   UNQ Nil;
```

Again, the compiler is able to deduce the full type itself. For clarity we advise always to specify the left-hand-side completely. The advised definition is:

```
::   UNQ List x     ->   Cons x (List x)
                    ->   Nil;
```

Example: Consider the following definition of a list containing unique elements:

```
::   List UNQ x     ->   Cons UNQ x (List UNQ x)
                    ->   Nil;
```

Due to the UNQ propagation rule the uniqueness of list elements will propagate over the Cons. As in the case above, the root of both alternatives as well as the algebraic type itself becomes unique and therefore the whole spine. So the complete list has to be unique (lists elements as well as the spine). The full definition is:

```
::   UNQ List UNQ x      ->   UNQ Cons UNQ x UNQ (List UNQ x)
                         ->   UNQ Nil;
```

The advised definition is:

```
::   UNQ List UNQ x      ->   Cons UNQ x (List UNQ x)
                         ->   Nil;
```

Unique types can be used as instantiations of polymorphic types. The meaning of a type T parameterized with a unique type is rather obvious: the resulting type is the same as the type that is obtained when the parameters (including the UNQ attributes) are substituted in the definition of T.

Take for example the following common definition of a list:

```
::   List x    ->   Cons x (List x)
               ->   Nil;
```

Then the List UNQ INT is internally expanded to the following full type definition:

```
::   UNQ List UNQ INT    ->   UNQ Cons UNQ INT UNQ (List UNQ INT)
                         ->   UNQ Nil;
```

The substitution of a UNQ instantiation in a polymorphic type will not always yield a UNQ type as result.

Take for example the following definition:

```
:: T x -> C (=> x (=> INT INT));
```

Then the type T UNQ INT is internally expanded to the following full type definition:

```
:: T UNQ INT -> C (=> UNQ INT UNQ (=> INT INT));
```

Since the propagation for function types propagates from left-to-right, T itself does not become unique.

Abstract types

Abstract types only appear in definition modules. They are specified by giving the full type specification of the left-hand-side of the corresponding type definition in the implementation module.

Example of the abstract version of the previous List type which was fully unique:

```
ABSTYPE
:: UNQ List UNQ x;
```

3.7 Untyped Clean

Full specification of types is recommended to avoid errors and to obtain efficient executables. However, the syntax rules show that one is not obligated to specify the type of the rules. It is even possible to turn type checking off. This means that types that are specified, are not checked but assumed to be correct. One restriction is imposed on untyped programs: such a program may not contain any tuples. However, this is not a serious drawback since the programmer can easily define tuples himself by means of an appropriate algebraic data type definition. Furthermore if the type of a function is not specified, it is assumed to be the most general type. So it is up to the programmer to make sure that such a function is applied in a correct manner and that its definition is correct.

For instance, consider the following GRS:

```
RULE
     Fac 0     ->   1;
     Fac n     ->   * n (Fac (-- n));
```

This untyped rule is assumed to be of type

```
::   Fac x     ->   y;
```

This facility to use Concurrent Clean in an untyped way is included to experiment with GRS rules that are not typable in the Milner system. Also if Concurrent Clean is used as intermediate language for an untyped language such as SASL (Turner (1979)), it is necessary that the types can be left out. Of course, it is very dangerous to use untyped rules. Furthermore, untyped rules will generally lead to very inefficient code.

Basic type matching

Severe problems will arise in untyped functions when predefined delta rules are being applied with arguments of a wrong type. A facility called basic type matching is provided to assist in run-time type checking. This facility makes it possible to specify a basic type as pattern.

The application of predefined rules with wrongly typed arguments can be prohibited with basic type matching:

```
RULE
     Fac 0          ->   1;
     Fac n:INT      ->   * n (Fac (-- n));
```

Basic type matching is also particularly useful for writing generic functions.

The definition of a generic function for addition using run-time type checking on basic types:

```
RULE
     Add x:INT  y:INT   ->   +  x y;
     Add x:REAL y:REAL  ->   +R x y;
     Add x:REAL y:INT   ->   +R x (ITOR y);
     Add x:INT  y:REAL  ->   +R (ITOR x) y;
     Add x      y       ->   ABORT "Type error !";
```

Of course, basic type matching only makes sense if compile-time type checking has been turned off. Such a run-time type check is possible on basic types only (`FILE` excluded). It is not possible to check whether or not an argument is of a particular user-defined type.

4 Modules

For a practical useful language separate compilation is a must. A module structure enables this with the facility to hide actual implementations of types and functions. This can be used to specify abstract data types (see also section 3.4). A Concurrent Clean program can therefore be split in several modules.

The syntax rules for modules:

```
CleanProgram   =    {Module}
               ;
Module         =    DefModule
               |    ImplModule
               ;
DefModule      =    'DEFINITION'   'MODULE' ModuleId ';' {Def}
               |     'SYSTEM'      'MODULE' ModuleId ';' {Def}
               ;
ImplModule     =    ['IMPLEMENTATION'] 'MODULE' ModuleId ';'
                         {Impl}
               ;
Def            =    Import
               |    Type-Block
               |    AbsType-Block
               |    DefRule-Block
               |    Macro-Block
               ;
Impl           =    Import
               |    Type-Block
               |    ImplRule-Block
               |    Macro-Block
               ;
```

A Concurrent Clean program consists of *definition modules* and *implementation modules*. Each module has to be put in a separate file. An implementation module consists of a set of macro, type and rule definitions. These definitions have only a *local scope*: they only have meaning in the implementation module they are defined in. However, with the use of a corresponding definition module, definitions can be *exported* to other modules. A definition module consists only of a set of type rules for exported types and for exported functions. Via an *import* statement such an exported definition can be used in another implementation or definition module.

A symbol S is said to be *defined* in a module M if it appears as root symbol of a rewrite rule, or if it appears in a type definition. A symbol S is said to be *declared* in a module M if it either is imported from another module or is defined in M itself. A symbol S in a module M may be defined only once. If a symbol S is defined in M it cannot be imported from another module. A symbol S can be imported more than once only if those imports refer to the same definition.

A module M1 *depends on* a module M2 if there exists a symbol S declared in M1 that is imported from M2. Cyclic dependencies of definition modules are prohibited, i.e. if a definition module M1 depends on another definition module M2 then M2 may not depend on M1.

In general, each definition module has a corresponding implementation module. An exception is the *main module*, which only consists of an implementation module. An alternative kind of definition module is the *system module*. System modules allow the corresponding implementation module to be not a Concurrent Clean program. System modules are used to implement predefined delta-rules efficiently.

4.1 Exporting

The types and rules defined in an implementation module have only a *local scope*: they only have meaning in the implementation module they are defined in. With the use of a corresponding definition module, definitions can be *exported* to other modules.

A symbol S defined in an implementation module is *exported* if that symbol is also defined in the corresponding definition module. Both rewrite rules and type definitions (including constructors) can be exported in this way.

A rewrite rule defined in an implementation module is *exported* if its type is defined in the corresponding definition module also. The type in the definition module has to be exactly the same as the type in the implementation module, including annotations and attributes. The types of the arguments and of the result of the function symbol must be declared in the definition module.

Types defined in an implementation module are *exported* if they are also defined in the definition module. For an abstract data type only the name of the type is exported but not its definition. In a definition module an abstract type is indicated by specifying the left-hand-side graph of its type rule. In the corresponding implementation module the abstract type is defined either as an algebraic type or as a synonym type.

4.2 Importing

To import types or rules one can use an import statement in which the name of the module to import from and the name of the function or type to be imported is stated. Imported types and rules can be regarded as being predefined. The syntax of import statements is as follows:

```
Import    =    'IMPORT'  ModuleId {',' ModuleId } ';'
          |    'FROM'    ModuleId 'IMPORT' Symbol {',' Symbol } ';'
```

There are two kinds of import statements. *Explicit* imports are import statements in which every type or rule that has to be imported from a particular module is explicitly mentioned. In this way only these explicitly indicated types and rules are declared within the importing module. *Implicit* imports are import statements in which only the module name to import from is mentioned. In this case all symbols that are declared in the imported module are imported (and declared). Note that not only all symbols exported by the indicated module are declared, but also all symbols that on their turn are imported in that definition module. In this case with one simple `IMPORT` statement all related declarations are imported implicitly. So definition modules can serve as a kind of 'pass-through'. For example, all (arithmetic) delta rules can be imported easily by importing the module 'delta':

```
DEFINITION MODULE delta;

IMPORT
     deltaB,   == Booleans
     deltaC,   == Characters
     deltaI,   == Integers
     deltaM,   == Miscellaneous
     deltaR,   == Reals
     deltaS;   == Strings
```

Import restrictions for types

An algebraic type symbol can only be imported explicitly together with its constructors, i.e. if a module M imports an algebraic type T then all the constructors of T (the right-hand-side root symbols of all the alternatives of the type rule for T) should also be imported. Note that when such a symbol is imported implicitly via an `IMPORT` statement this is always the case. For a synonym type the same restriction holds as for an algebraic type. So, if one imports a synonym type T one should also import the type T2 for which T is a synonym. An imported abstract data type is comparable with a predefined basic type of which the internal structure remains hidden. Therefore, no further imports are needed.

Import restrictions for rules

A function symbol can only be imported explicitly in a module if the types of its arguments and the type of its result have also been declared (imported) in that module.

4.3 Definition modules

In a definition module it is generally defined which of the types and rules defined in the corresponding implementation module are made visible to the outside world. All other definitions in the implementation module remain hidden.

Example of a definition module:

```
DEFINITION MODULE ListOperations;

ABSTYPE
:: List x     ;

RULE
:: Hd     (List x)  ->   x;        ==   take the head of a list
:: Tl     (List x)  ->   List x;   ==   take the tail of a list
:: IsNil  (List x)  ->   BOOL;     ==   is the list empty ?
```

4.4 Implementation modules

An implementation module and a definition module *correspond* to each other if the names of the two modules are the same. A definition module can have at most one corresponding implementation module. An implementation module that is not the main module, must have exactly one corresponding definition module.

An implementation module must at least contain the definitions of the exported rewrite rules and types defined in the definition module. The implementations of the abstract data types that are defined in the corresponding definition module, are given by defining them in the implementation module as an algebraic or synonym type. For the rewrite rules that are defined in the corresponding definition module, a rule definition is given in the implementation module.

An implementation module can also contain auxiliary definitions of types and rules that are not visible to the outside world.

Example of an implementation module that could be the implementation module corresponding to the definition module of the previous example:

```
IMPLEMENTATION MODULE ListOperations;

FROM deltaM IMPORT ABORT;

TYPE
:: List x -> [x];

RULE
:: Hd (List x) -> x;
   Hd [x | y]  -> x;
   Hd x        -> ABORT "head of empty list";

:: Tl (List x) -> List x;
   Tl [x | y]  -> y;
   Tl x        -> ABORT "tail of empty list";

:: IsNil (List x) -> BOOL;
   IsNil []       -> TRUE;
   IsNil x        -> FALSE;
```

Main or Start module

An implementation module M can be recognised as being the *main* module or *start* module because in the module header of M the keyword `IMPLEMENTATION` has been left out. The main module has no corresponding definition module. The main module should always contain a declaration of the `Start` rule, i.e. this rule should either be defined or imported from another module.

4.5 System modules

System modules are special definition modules indicated by the keyword `SYSTEM` instead of `DEFINITION`. The corresponding implementation module does not contain ordinary rules, but abstract PABC-machine code instead. This facility makes it possible to program directly in PABC-code, in particular in order to increase execution speed of heavily used functions or complex data structures. At demand, the Concurrent Clean compiler will substitute the PABC code of a function *in-line* at the place in which the corresponding function is called. In this way execution speed can be increased even more.

Typically, predefined delta rules for arithmetic and file I/O are implemented as system modules. System modules have as advantage that a certain increase of efficiency can be obtained on a device independent manner, since PABC-code is device independent. It goes without saying that for PABC programming advanced knowledge is needed of the code generation schemes and the run-time behaviour of the PABC-machine. Imprudent programming of system modules will generally lead to unpredictable errors. For a summary of the PABC code the reader is referred to (Nöcker (1989)).

5 Input/output handling

Incorporating input/output (I/O) in functional languages, causes a number of well-known problems. The two major conceptual problems are the need for side-effects and the need for precise specification of evaluation order in I/O systems. It is difficult to realise these satisfactory in a functional model which doesn't have the notion of side-effects and lacks the control structures that seem to be necessary when specifying I/O. The solution for these problems that has been implemented in Concurrent Clean makes it possible to program sophisticated I/O features (ranging from file I/O to window-based interactions) in a very elegant way.

5.1 Functional I/O

The solution* to the problems with functional I/O that is implemented for Concurrent Clean (Achten (1992,1993)) is based on a kind of explicit environment passing scheme and guaranteed single threadedness using the `UNQ` type attribute of section 3.6.

Explicit environment passing where needed

One of the solutions for functional I/O is to pass a special predefined environment as a parameter to functions. I/O is always performed on such an environment object (for example a file). Existing environment passing schemes either pass the environment to *all* functions (explicit environment passing schemes) or to *none* (implicit environment passing schemes). An important characteristic of those methods is the *monolithic* nature of the environment object: There is only *one* such object on which *all* I/O operations must be performed.

The following example illustrates the principle of environment passing by means of a function that writes a list of strings to a file. In this case the environment is a Concurrent Clean `FILE`. The function `FWriteS` returns a new `FILE` environment on which the `FWriteStrings` function can continue recursively. When all strings have been written to the file the resulting `FILE` is returned.

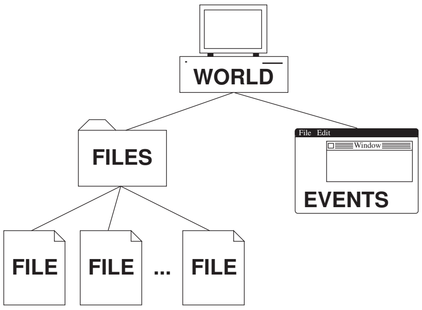
```
::   FWriteStrings [STRING] FILE -> FILE;
     FWriteStrings [string|rest] file
     -> FWriteStrings rest (FWriteS string file);
     FWriteStrings [] file -> file;
```

In Concurrent Clean a new type of explicit environment passing scheme is used, called *explicit environment passing where needed*. Rather than passing the environment to all functions the environment is passed only to functions with side-effects. The main advantage of this approach is that over-specification of evaluation order can be avoided: Only in those parts of the program in which the evaluation order is important (i.e. in which side-effects occur) the evaluation order can be controlled by means of the environment parameter.

The second refinement that has been made to the explicit environment passing scheme is that in Clean the environment consists of several sub-environments, each used for a particular class of I/O operations. The main advantage of this approach is that unrelated I/O objects can be handled independently. For example, two different files can be handled separately in different parts of the program. With a monolithic environment this is impossible: I/O objects are always related to each other then by the monolithic environment, even when in fact they have no relationship at all. The environments that are defined in Concurrent Clean are `WORLD`, `FILES`, `FILE` and `EVENTS`. The `WORLD` contains the other environments. There is at most one `WORLD` in a program, which is provided by the `Start` rule. From this `WORLD` the disjoint environments `FILES` and `EVENTS` can be retrieved. The `FILES` environment represents the state of the file system and gives access to all files (of type `FILE`) visible to the program. The `EVENTS` environment

* The Concurrent Clean System only supports the I/O facilities when concrete code is generated. In the simulator the very limited I/O operations offered by the `deltaIO` library (see appendix B) can be used to write interactive programs.

is used to do so-called event I/O: it makes it possible to create and manipulate menus, windows, dialogs etcetera.



The environment hierarchy of Concurrent Clean.

Single threaded environments

For efficiency reasons one would like to have a one-to-one correspondence between abstract, functional objects used in the language and the concrete objects used by the operating system: reading or writing from a `FILE` causes direct reading or writing from a real file. However, a naive implementation may conflict with the functional semantics by not effectively removing the side-effects, which may for example result in the possibility that a file is used as a global variable. The following example illustrates the conflicts that may arise:

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

When `FWriteC` instantaneously writes the character to the file the wrong result `(file++"a", file++"a")` will be returned by this function. This problem is caused by the fact that the file is shared in the right-hand-side of `F`: the file is not used single threadedly. Only when an object is used single threadedly destructive updates (such as instantaneous writing to a file) can be allowed. In section 3.6 a method to enable compile-time checking of single threadedness (uniqueness) has been introduced. With the use of the `UNQ` type attribute presented there it can be checked whether objects are used single threadedly. Functions that destructively update an object therefore always expect this object (environment) to be `UNQ` attributed. For example, `FWriteC` is of type `=> CHAR (=> UNQ FILE UNQ FILE)`. Therefore the function `F` in the example above is of type `=> UNQ FILE (UNQ FILE, UNQ FILE)`. Because of the `UNQ` type attributes the compiler would reject this definition of `F`. In the `FWriteStrings` example above the file is used single threadedly, but its type should be:

```
::   FWriteStrings [STRING] UNQ FILE -> UNQ FILE;
```

In Concurrent Clean the environments `FILES` and `EVENTS` are always `UNQ`. A `FILE` must be `UNQ` attributed when one wants to write to it. When a file is used read-only there is no need to use it single threadedly, so the `UNQ` attribute is not obligatory then. The `WORLD` environment mus be `UNQ` attributed in the `Start` rule.

I/O in Concurrent Clean

The concepts described above are used in Concurrent Clean to implement functional I/O. The environments `WORLD` and `FILE` are predefined as basic types. The environments `FILES` and `EVENTS` are defined in the I/O libraries as `UNQ` abstract types. On the (`UNQ`) `WORLD` environment functions have been defined to open and close (retrieve and put back) the sub-environments `FILES` and `EVENTS`, whereas (`UNQ`) `FILE`s can again be opened and closed via the `FILES` environment by means of other predefined functions. For `FILE`s and `UNQ FILE`s the file I/O library has been implemented. To be able to do window-based I/O (hereafter called *event I/O*) the event I/O libraries have been implemented which contain types and functions for the creation and manipulation of graphical interface objects such as windows, dialogs and menus. Because all functions in the I/O libraries that have a side-effect work on a unique environment parameter these side-effects are safe: they do not violate the functional semantics. In the following sections the I/O libraries will be examined more closely. A complete overview of the functions defined in the I/O libraries can be found in appendix C.

5.2 File I/O

In Concurrent Clean files can only be opened and closed through the `FILES` environment. The `FILES` environment gives access to all files visible to the program. The library module `deltaFile` contains the following functions to open and close the `FILES` environment:

```
::   OpenFiles  UNQ WORLD -> (FILES, UNQ WORLD);
::   CloseFiles FILES UNQ WORLD -> UNQ WORLD;
```

Attempts to open the `FILES` environment when it is already open result in a run-time error. This, and the uniqueness of the `FILES` environment guarantees that there is always at most one `FILES` environment in a program.

As an example of the use of the `FILES` environment the `Start` rule of a program that copies a file called "source" to a file called "dest" is given. The function that does the actual copying (`CopyFile`) is given later on.

```
::   Start UNQ WORLD -> UNQ WORLD;
     Start world -> CloseFiles files' world',
          files'          : CopyFile "source" "dest" files,
          (files, world'): OpenFiles world;
```

Opening and closing files

All predefined functions working on files are defined in the module `deltaFile`. For the opening of (`UNQ`) `FILE`s the following predefined functions exist in that module:

```
::   FOpen    STRING INT FILES -> (BOOL, UNQ FILE, FILES);
::   SFOpen   STRING INT FILES -> (BOOL, FILE    , FILES);
```

The `STRING` parameter should contain the complete pathname of the file to be opened. When the pathname is just a filename the file will be opened in the current working directory. The library module `deltaSystem` contains functions that prefix a filename with the complete pathname of the 'home' directory or the 'application' directory. The 'application' directory is the directory in which the application resides. The 'home' directory is the home directory of the user. On single-user operating systems the 'home' directory will be the current working directory.

The second parameter of the `(S)FOpen` function is the file mode, which indicates whether the file should be opened for reading or writing and whether it should be considered as a data or as a text file. Six different file modes have been predefined. The boolean output parameter reports whether the operation succeeded. `FOpen` returns a `UNQ FILE` which can be read and/or written, depending on the file mode. When the file that is opened by `FOpen` was already open a run-time error will be generated, because otherwise single threaded use of `UNQ FILE`s can not be guaranteed. `SFOpen` (Shared File Open) returns a (non-unique) `FILE`, which may be opened more than once without closing it. `FILE`s can be used multi-threadedly, which is no problem because it is impossible to write to `FILE`s. No write functions have been defined on `FILE`s, just a set of read functions. With the `FShare` function `UNQ FILE`s can explicitly be converted to `FILE`s.

The possibility to close files is important for two reasons. Firstly `FOpen` results in a run-time error when the file to be opened was already open, which means that it would be necessary to keep track of all open files. When it is possible to close files this is no longer necessary. Secondly most operating systems impose a limit on the number of files that can be open at one time. Without the possibility to close files explicitly there is no way to minimise the number of files that is open. The following predefined functions can be used to close (`UNQ`) `FILE`s:

```
::   FClose  UNQ FILE FILES   -> (BOOL, FILES);
::   SFClose FILE     FILES   -> (BOOL, FILES);
```

The boolean output parameter reports whether closing of the file succeeded. For UNQ FILEs a special function exists to change the mode of the file without having to close the file explicitly:

```
::   FReOpen UNQ FILE INT     -> (BOOL, UNQ FILE);
```

This function does not need the FILES environment as parameter which makes it a lot easier to use than a combination of FClose and FOpen.

An example of the use of the open and close functions for files is the CopyFile function promised earlier. It takes the path name of two files and the FILES environment as parameters and copies the first file to the second. The function that does the actual copying (CopyFileContent) will be given later on. Tests whether the opening and closing of the files succeeded (inspection of the booleans suc1 - suc4) have been left out.

```
::   CopyFile STRING STRING FILES -> FILES;
     CopyFile sname dname files -> files4,
          (suc4, files4)        : SFClose source files3,
          (suc3, files3)        : FClose  dest'  files2,
          dest'                 : CopyFileContent source dest,
          (suc2, source, files2): SFOpen sname FReadData  files1,
          (suc1, dest  , files1): FOpen  dname FWriteData files;
```

Because it is certain that the source file will only be read it can be opened with SFOpen. The destination file has to be opened as a UNQ FILE using FOpen.

Reading from files

Functions have been defined to read CHARs, INTs, REALs and STRINGs from a file. These functions work on text files as well as data files, except for the FReadLine function, that reads a line (ended by a newline character) from a text file. When FReadLine is applied on a file that is opened in data mode a run-time error will be generated. When one of the other functions is applied to a text file they will try to read the textual representation of the object they should return. In the case of integers and reals first tabs and spaces are skipped, after which the textual representation is read and converted into an integer or a real. When these functions are applied to data files they will read a number of bytes from that file, depending on the internal representation of the object they should return. For example, FReadC reads one byte (a Clean CHAR) from a data file. All functions that read from files return a new UNQ FILE on which reading can continue. Furthermore often a boolean is returned to report success or failure of the read operation. FReadS and FReadLine report failure by returning an empty string.

Apart from the read operations that work on UNQ FILEs the same set of functions has been defined for FILEs. These functions all begin with SF instead of F. They are exactly the same as the other read functions, but they only work on non-unique files, for instance files that have been opened with SFOpen or that have been converted by means of FShare.

Writing to files

Functions have been defined to write characters, integers, reals or strings to files. These functions have as arguments the object to be written and a UNQ FILE. They return a new UNQ FILE on which writing can continue. When objects are written to text files their textual representation is written to the file. For example, when the integer 102 is written to a text file in fact the characters ‘1’, ‘0’ and ‘2’ are written to that file. When objects are written to a data file, a certain amount of bytes containing the value of the object is written to the file. The amount of bytes that is written to the file depends on the internal representation of the object. The write functions do not return a boolean reporting success or failure. A special function FError has been implemented that returns a boolean indicating whether or not an error has occurred during previous file I/O actions (TRUE means that an error has occurred, FALSE means no error).

Other functions on files

Apart from functions that read from or write to files, some other useful functions have been implemented. The `FEnd` function tests for end of file on `UNQ FILE`s. It returns a boolean and a new unique file. There is also a `SFEnd` function that does the same for non-unique `FILE`s. This function only returns a boolean, because other operations can always continue on a copy of the `FILE`. In guards the `SFEnd` function can also be used on `UNQ FILE`s, because in guards the `FILE` looses its `UNQ` attribute (see section 3.6).

The `FPosition` function returns the position of the file pointer of an `UNQ FILE` as an integer indicating the number of bytes between the beginning of the file and current file pointer. It of course also returns a new `UNQ FILE`. The position can be used as parameter of the seek functions. There also is a `SFPosition` function, that only works on `FILE`s and that does not return a continuation.

With the `FSeek` function the file pointer can be moved explicitly. It has a `UNQ FILE`, an offset (an integer) and a seek mode as parameters. The seek mode can be `FSeekSet` (the file pointer will be set offset bytes from the beginning of the file), `FSeekCur` (the file pointer will move offset bytes from its current position), or `FSeekEnd` (the file pointer will move offset bytes from the end of the file). There is also a `SFSeek` function, which does the same, but that only works on non-unique `FILE`s.

The following function uses the seek and position functions to calculate the size of a file (in bytes):

```
::   SFSize FILE -> INT;
     SFSize file -> SFPosition file',
          (s,file'): SFSeek file 0 FSeekEnd;
```

First the file pointer is moved to the end of the file by means of the `SFSeek` function, after which the position of the file pointer (i.e. the number of bytes in the file) is returned by `SFPosition`.

An example of file I/O

As an example of Concurrent Clean's file I/O the complete program is given that copies the contents of a file called "source" to a file called "dest". The `Start` rule and the `CopyFile` function defined earlier have been combined into one `Start` rule. It is not obligatory to close environments such as `FILE` or `FILES` in Concurrent Clean, so the closing of these environments has been left out. This makes the `Start` rule much simpler:

```
MODULE copyfile;

IMPORT deltaFile;

RULE
::   Start UNQ WORLD -> UNQ FILE;
     Start world -> CopyFileContent source dest,
          (s2, source, files2): SFOpen "source" FReadData  files1,
          (s1, dest  , files1): FOpen  "dest"   FWriteData files,
          (files, world')     : OpenFiles world;

::   CopyFileContent FILE UNQ FILE -> UNQ FILE;
     CopyFileContent source dest
     ->   dest, IF SFEnd source
     ->   CopyFileContent source' dest',
          dest'                   : FWriteC char dest,
          (success, char, source'): SFReadC source;
```

In this program the `WORLD` and `FILES` environments become garbage in the `Start` rule, which is perfectly safe, because they are not needed any more. In more complicated programs one must be careful: when an environment has become garbage it can never be accessed again. For example, when the `FILES` environment becomes garbage it is still open, which means it cannot be retrieved any more from the `WORLD` by means of `OpenFiles`. This means that except for the `FILE`s that were already open no file access is possible any more. The same holds for the other unique environments (`UNQ WORLD`, `UNQ FILE` and `EVENTS`).

5.3 Event I/O

Event I/O is a different class of I/O than file I/O. In event I/O the objects that are being manipulated are graphical interface objects such as windows, menus and dialogs. Graphical interface systems operate event-driven: Actions performed by the user generate events to the program. The operating system also generates events, to notify the program that things have changed. Programming an interactive application comes down to one problem: How should the program respond to the events given to it? To solve this problem appears to be a very complicated task.

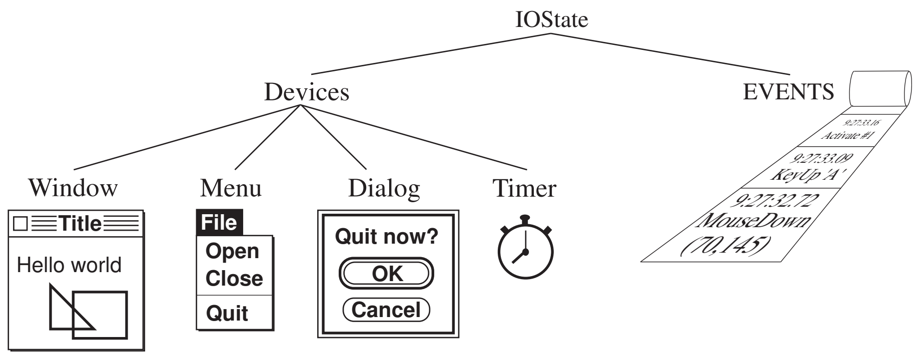
In the Concurrent Clean event I/O library it is tried to simplify this problem as much as possible by offering a way to define graphical interface objects on a level that is very close to the way these objects actually appear on the screen. The basic principle of Concurrent Clean event I/O is that parameterized constructors of well chosen predefined algebraic types are used to define graphical interface objects. These object definitions are, among other things, parameterized with so-called *event handlers*: program defined functions that specify the response of the corresponding interface object to the events given to it. The event I/O library completely takes care of the task of calling the right event handler for each event: the programmer only specifies *what* should be done, not *how* this should be done.

In Concurrent Clean the set of available interface objects is structured by means of devices. A *device* is a dynamic set of related interface objects such as windows or menus. There are four devices: the menu device, the window device, the dialog device and the timer device. The behaviour of a device is partially fixed by the system and is partially defined by the program. The part that is fixed by the system concerns all tasks that can be handled automatically, such as highlighting of menus and scrolling of windows.

Interactions

The concept of events and devices can now be used to define *interactions*. An interaction is a state transition system in which I/O is performed. State transition systems are in general defined by an *initial state*, a set of all possible *transitions* and a *final state*. Here we transfer these concepts to interactions.

The *state* of an interaction is composed of two objects: the *program state* and the *I/O state*. The program state is a program defined data structure of an arbitrary `UNQ` type. It is used to keep all global data needed during the interaction. The I/O state is modelled by the `IOState` environment. The `IOState` environment is an `UNQ` abstract object that contains the event stream (modelled by the `EVENTS` environment) and the devices that participate in the interaction:



The predefined functions in the event I/O library work on an environment of type `IOState`, rather than on the `EVENTS` environment. In this way it is made explicit that all event handling is done by the library.

The *initial state* of an interaction consists of the initial program state, a definition of the devices that participate in the interaction and the event stream (`EVENTS`). The devices and the `EVENTS` are combined to create the initial `IOState` environment. The event stream can be retrieved from and put back into the `WORLD` by means of the following predefined functions from the `deltaEventIO` module:

```
::   OpenEvents UNQ WORLD -> (EVENTS, UNQ WORLD);
::   CloseEvents EVENTS UNQ WORLD -> UNQ WORLD;
```

The initial program state and the device definitions are of course provided by the program.

The *transition set* of an interaction is the collection of event handlers of the devices contained in the `IOState`. Transitions are triggered by the events in the event stream. Each event in the event stream is dispatched to the proper device, which computes the next interaction state by calling the proper event handler. Event handlers generally have the type: `=> s (=> (IOState s) (s, IOState s)`. The order of evaluation is such that it is guaranteed that each subsequent transition is reduced only after the previous transition has yielded a complete (normal form) `IOState`.

Interactions are started by the following predefined rule from the `deltaEventIO` module:

```
::   StartIO (IOSystem s) s [InitialIO s] EVENTS -> (s, EVENTS);
```

In its arguments the state transition elements presented above can be identified. The first parameter (`s`) serves as the initial program state. The `IOSystem` parameter contains a list of device definitions. Together with the `EVENTS` environment this parameter is used to create the initial `IOState`. As mentioned, the transition set is defined by the event handlers of the devices. This means that all parts of the interaction are present and that the interaction can be started. Before the interaction is started the `InitialIO` functions are evaluated. These are functions of type `=> s (=> (IOState s) (s, IOState s))`, with which the initial settings of the interaction can be defined.

Interactions are handled recursively by `StartIO` until they reach their *final state*. The final state of an interaction is reached when one of the program defined event handlers quits the interaction. The interaction can be quitted by applying the predefined rule `QuitIO` from the `deltaEventIO` module to the current `IOState`:

```
::   QuitIO (IOState s) -> IOState s;
```

`QuitIO` removes all devices from the `IOState`, which causes `StartIO` to terminate with the final interaction state, formed by the current program state and the resulting event stream (`EVENTS`), from which the events that have been handled during the interaction have been removed.

It is also possible to define *nested* interactions. A nested interaction is an interaction that temporarily replaces another interaction. This is useful for example when one wants to call a text editor in a programming environment. A nested interaction is started by the following predefined function:

```
::   NestIO (IOSystem t) t [InitialIO t] (IOState s)
                                           -> (t, IOState s);
```

The current interaction is represented by the `IOState` argument. The devices of this interaction will be hidden, such that all windows, menus and dialogs of the current interaction will become invisible. With the event stream from the current `IOState` and the devices specified in the `IOSystem` argument a new initial `IOState` is created. The initial program state of the nested interaction is defined by the first argument (`t`). Just like `StartIO`, `NestIO` is terminated by `QuitIO`. When the nested interaction is quitted `NestIO` returns the final program state of the nested interaction and the old `IOState`, such that the original interaction re-appears. Interactions may be nested arbitrarily deep.

Without predefining special functions it is possible to define sequences of interactions: one interaction after the other. Suppose `S` and `T` are types of program states. A sequence of two interactions, each with its own program state and devices, can be defined by means of the following function:

```
:: Sequence S (IOSystem S) (=> S T) (IOSystem T) EVENTS
   ->  (T, EVENTS);
   Sequence s iosystems stot iosystemt events
   ->  StartIO iosystemt (stot s') [] events',
       (s', events'): StartIO iosystems s [] events;
```

The resulting program state of the first interaction (`s`) is used by the function `stot` to create the initial program state of the second interaction. The devices of the interactions are defined by the `IOSystem` arguments (`iosystems` and `iosystemt`). When the first interaction is quitted, the resulting program state and event stream are passed from the first interaction to the second. When the second interaction is quitted, the function terminates with the final program state of the second interaction and the resulting event stream. Of course, it is also possible to create sequences of more than two interactions.

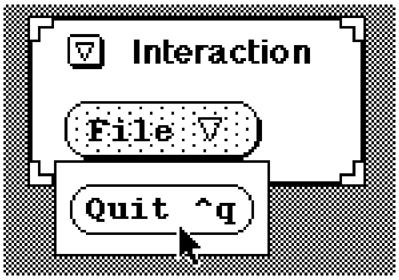
An example of event I/O

As an example of the way event I/O is done in Concurrent Clean a program is given that defines one menu, containing one menu item (Quit). When this item is selected the program terminates. In this example predefined constructors are set bold, predefined functions and types are set in italics and event handlers are set bold and in italics. In the remainder of this chapter the same conventions will be used.

```
MODULE Interaction;

IMPORT deltaEventIO;

TYPE
::   UNQ S -> INT;     == The program state

MACRO
     FileId -> 1;     QuitId -> 2;

RULE
::   Start UNQ WORLD -> UNQ WORLD;
     Start world -> CloseEvents events' world',
          (ps', events'): StartIO [MenuSystem [menu]] 0 [] events,
          (events, world'): OpenEvents world,
          menu: PullDownMenu FileId "File" Able
                  [MenuItem QuitId "Quit" (Key 'Q') Able Quit];

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

The only device participating in this interaction is the menu device. It is defined by means of the `MenuSystem` constructor, which has a list of `PullDownMenus` as parameter. A pull down menu contains menu items. A `MenuItem` is parameterized with its id, title, key shortcut, selectability and of course its event handler.

The Clean event I/O libraries have been implemented for several types of operating systems. These implementations completely preserve the look and feel of the corresponding interface systems. The following picture shows how the example looks on an Apple Macintosh and on an X Windows / Open Look system:



The event I/O library

In the previous section the basic principles of Concurrent Clean event I/O have been explained. But, nothing has been said yet about the construction and manipulation of graphical interface objects in Concurrent Clean. The subsequent sections intend to do so by taking a closer look at the types and functions that are predefined in the event I/O library. The predefined types for the constuction of interface objects can be found in the module `deltaIOSystem`, whereas the functions used to manipulate the interface objects during the interaction can be found in the other modules of the event I/O library (see appendix C).

As described earlier, the initial `IOState` is supplied with devices in the `StartIO` rule. The device definitions are collected in a data structure called `IOSystem`:

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

Each device system has a list of corresponding interface object definitions as parameter. In the following sections the definition of each of the devices is discussed.

5.4 Menus

The menu device contains all pull down menus used in the interaction. All menus have to be defined at the beginning of the interaction, i.e. in the `IOSystem` of the `StartIO` rule. A menu consists of menu items: commands or options from which the user can choose. Menu items are selected with the mouse or with a special key combination that is assigned to a it (a so-called key shortcut). The menus of the menu device usually appear in a menu bar at the top of the screen, or in a special application window, in the same order as they are specified in the list of `MenuDefs`. On some interface systems the complete menu system is also available as a pop up menu in each window of the application. A pull down menu is specified in Clean by the predefined algebraic type `MenuDef`:

```
::   MenuDef UNQ s UNQ io -> PullDownMenu MenuId MenuTitle
                                  SelectState [MenuElement s io];
::   MenuId      -> INT;
::   MenuTitle   -> STRING;
::   SelectState -> Able | Unable;
```

A `PullDownMenu` is defined by the following parameters:

- `MenuId`: An integer uniquely identifying this menu. In Concurrent Clean all interface objects have such an id, which is used to refer to the object during the interaction. Id's must be greater than zero and must be unique within a class of interface objects. This means that two menus must have different id's, but that for example a menu may have the same id as a window or a dialog.
- `MenuTitle`: The title of the menu. On the screen menus can be identified by their title.
- `SelectState`: Indicates whether the menu is initially selectable (`Able`) or not (`Unable`). Several interface objects have a `SelectState` as parameter. Usually the selectability of an interface object has an effect on its appearance on the screen (disabled items often appear dimmed in some way).
- A list of `MenuElements`: In this list the menu items are defined. We will take a closer look at the way menu items are defined in the next section. Menu items appear in the menu in the same order as they are specified in the list.

Although all menus have to be defined at the beginning of the interaction, they can be changed during the course of the interaction. The library module `deltaMenu` contains functions to enable and disable menus and menu items, mark and unmark menu items and to change the title and functionality of menu items. Furthermore, certain types of items can be removed from or inserted into menus during the interaction.

Menu items

For menu items the algebraic type `MenuElement` has been predefined:

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

As can be seen six kinds of menu items have been predefined:

`MenuItems` are 'standard' menu items, which can be selected with the mouse or with a `KeyShortcut`. When a menu item is selected the corresponding event handler (`MenuFunction`) is called.

A `CheckMenuItem` can be marked with a checkmark. The `MarkState` parameter indicates whether the item is initially marked.

A `SubMenuItem` defines a submenu. When the related item is selected by the user a new menu appears in which further selection can be done. The items of the submenu are defined by the list of `MenuElements`.

A `MenuItemGroup` is a group of menu items from which items can be removed and to which items can be added during the interaction. The `MenuElements` in a `MenuItemGroup` can only be `MenuItems`, `CheckMenuItems` and `MenuSeparators`. When other kinds of menu items are put into an `MenuItemGroup` they are simply ignored.

A group of `MenuRadioItems` is a group of menu items (options) of which exactly one is selected at a time. The selected item will be recognisable for the user in a system dependent way. The `MenuItemId` of the `MenuRadioItems` constructor is the id of the item that is selected initially. When this id cannot be found in the list of radio items the first item in the list will be selected.

A `MenuSeparator` separates two menu items, for example by means of a horizontal line, or by means of an empty, non-selectable menu item.

An example of a menu definition

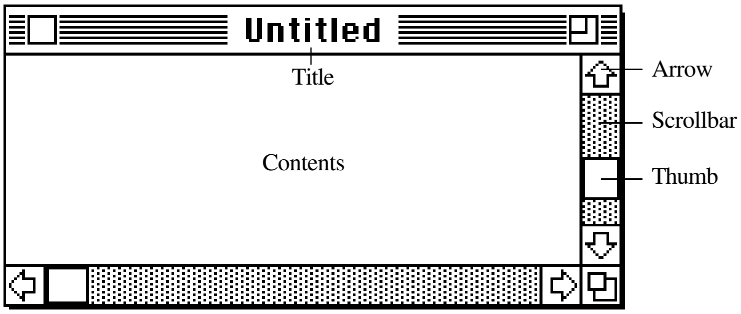
The menu defined in the following example contains three menu items. The third item (Quit) quits the interaction. The first item (Disable) disables the Quit item, the second item (Enable) enables the Quit item. The Quit item is separated from the other two items by means of a menu separator.

```
::   MenuExample -> MenuDef s (IOState s);
     MenuExample
     -> PullDownMenu 1 "Example" Able [
          MenuItem 1 "Disable" NoKey    Able DisableQuit,
          MenuItem 2 "Enable"  NoKey    Able EnableQuit,
          MenuSeparator,
          MenuItem 3 "Quit"   (Key 'Q') Able Quit ];

::   DisableQuit s (IOState s) -> (s, IOState s);
     DisableQuit s io -> (s, DisableMenuItems [3] io);

::   EnableQuit s (IOState s) -> (s, IOState s);
     EnableQuit s io -> (s, EnableMenuItems [3] io);
```

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

5.5 Windows

Windows are the basic medium through which interactive applications and users communicate. An application can have an arbitrary number of windows open. Of these windows at most one is *active*. The active window is the window to which all keyboard events are directed. Applications can display anything in a window: a window gives a view on an arbitrary picture. Windows are also used to structure user input to applications: a window may accept keyboard and/or mouse input. The following picture shows a typical (Macintosh) window and clarifies some of the terminology used in the rest of this section.



Windows can be opened at the beginning of an interaction by defining windows in the initial I/O system provided to the `StartIO` rule. During an interaction windows can be opened and closed by means of predefined functions from the `deltaWindow` module. In Concurrent Clean windows are defined by means of the predefined algebraic type `WindowDef`:

```
::   WindowDef UNQ s UNQ io
     -> ScrollWindow  WindowId WindowPos WindowTitle
                      ScrollBarDef ScrollBarDef PictureDomain
                      MinimumWindowSize InitialWindowSize
                      (UpdateFunction s) [WindowAttribute s io]
     -> FixedWindow   WindowId WindowPos WindowTitle PictureDomain
                      (UpdateFunction s) [WindowAttribute s io];
::   WindowId           -> INT;
::   WindowPos          -> (INT,INT);
::   WindowTitle        -> STRING;
::   ScrollBarDef       -> ScrollBar ThumbValue ScrollValue;
::   ThumbValue         -> Thumb INT;
::   ScrollValue        -> Scroll INT;
::   MinimumWindowSize  -> (INT,INT);
::   InitialWindowSize  -> (INT,INT);
::   UpdateFunction UNQ s
     -> => UpdateArea (=> s (s, [DrawFunction]);
::   UpdateArea         -> [Rectangle];
```

One can define two kinds of windows: `ScrollWindows` and `FixedWindows`. A `ScrollWindow` gives a view on a picture that may be bigger than the window itself. It has scroll bars to be able to scroll through the picture. Its size can be changed by the user. A `FixedWindow` has a fixed size that is determined by the size of its picture. It has no scroll bars and the size of the window cannot be changed by the user. When a `FixedWindow` is opened that is larger than the screen's dimensions it is automatically converted into a `ScrollWindow`. Functions that return the maximum size of `FixedWindows` and `ScrollWindows` can be found in the library module `deltaSystem`. A `ScrollWindow` is defined by the following parameters:

- WindowId: An integer uniquely identifying this window. When it is tried to open a window with the same id as a window that is already open, the window will not be opened. Instead the existing window with the matching id will become the active window.
- WindowPos: The initial position of the top-left corner of the window on the screen. When this position places the window outside the screen an acceptable position will be chosen by the system.
- WindowTitle: The title of the window.
- Two ScrollBarDefs: The first one defines the horizontal scroll bar, the second one the vertical scroll bar. A scroll bar is defined by the position of its Thumb, which defines which part of the window's picture is visible, and by its Scroll value, which defines how many pixels the window will scroll when one of the scroll bar's arrows is clicked. Both values must be between the bounds of the picture domain. When they are not they are automatically adjusted.
- PictureDomain: The top-left and bottom-right corner of the window's picture (in screen pixels). A run-time error will be generated when the top-left corner of the domain is not to the upper left of the bottom-right corner. The size of the window is limited by the size of the picture domain: the contents of the window can never become larger than the picture it shows. The PictureDomain specifies a coordinate system in which drawing takes place. The ranges of this coordinate system are limited by the draw functions, that usually cannot handle coordinates larger or smaller than certain values. The minimum and maximum values of the boundaries of the picture domain are defined in the module deltaSystem.
- MinimumWindowSize: The minimum width and height of the window (in pixels). The minimum window size is automatically adjusted to an acceptable value when it is either larger than the initial window size, or larger than the size of the screen, or larger than the size of the picture domain.
- InitialWindowSize: The initial width and height of the window. When this size is either larger than the size of the screen, or larger than the size of the picture domain, it is automatically adjusted to an acceptable value.
- UpdateFunction: The function that is called when (parts of) the window need to be redrawn. Its arguments are a list of Rectangles (UpdateArea) that should be updated and the program state. It returns the new program state and a list of DrawFunctions that will be applied to the picture of the window in order of appearance in the list. A DrawFunction is a function of type => Picture Picture. The types Rectangle and Picture and all kinds of draw functions are predefined in the module deltaPicture. It is guaranteed by the library that the update area is empty before the update tkaes place and that no drawing will take place outside the UpdateArea. The update function is also called when the window is opened, with the complete contents as UpdateArea.
- A list of WindowAttributes: This list may contain event handlers for all kinds of events that a window may accept: Mouse events, keyboard events, activate/deactivate events and request-to-close events. Furthermore the cursor shape (the shape of the mouse pointer) inside the window contents may be defined in this list. The next section gives a more complete overview of these attributes.

To clarify the above standing an example of a window definition is given, that will look exactly like the window shown in the picture given earlier:

```
::  MyWindow -> WindowDef s (IOState s);
    MyWindow -> ScrollWindow 1 (0,0) "Untitled"
                     ScrollBar (Thumb 0)   (Scroll 10)
                     ScrollBar (Thumb 120) (Scroll 10)
                     ((-100,-50),(300,200)) (50,50) (200,100)
                     UpdateMyWindow [];

::  UpdateMyWindow UpdateArea s -> (s, [DrawFunction]);
    UpdateMyWindow area s -> (s, [DrawString "Hello world"]);
```

Because the attribute list of the window is empty, the window does not respond to mouse or keyboard input. The update function ignores the update area passed to it and just draws "Hello world" in the window.

Window attributes

For window attributes the following algebraic type has been predefined:

```
::   WindowAttribute UNQ s UNQ io
     -> Activate    (WindowFunction s io)
     -> Deactivate  (WindowFunction s io)
     -> GoAway      (WindowFunction s io)
     -> Mouse       SelectState (MouseFunction s io)
     -> Keyboard    SelectState (KeyboardFunction s io)
     -> Cursor      CursorShape
     -> StandByWindow;
::   WindowFunction UNQ s UNQ io -> => s (=> io (s,io));
::   MouseFunction UNQ s UNQ io
     -> => MouseState (=> s (=> io (s,io)));
::   KeyboardFunction UNQ s UNQ io
     -> => KeyboardState (=> s (=> io (s,io)));
```

The definitions of `CursorShape`, `MouseState` and `KeyboardState` can be found in appendix C, module `deltaIOSystem`.

The event handler defined by means of the `Activate` constructor is called the activate function. It is called whenever a window becomes the active window. Usually, such an event requires no further action. But, for example, when you have defined a menu that indicates which of the open windows is the active window, this function can be very useful. When the activate function is called it is guaranteed that the corresponding window already is the active window. The `Deactivate` function is called when a window becomes inactive. It is not guaranteed that the corresponding window is still active at the moment that the deactivate function is called.

The `GoAway` function is called when the user requests to close the window. When no go-away function is defined by the program the window is simply closed. With the go-away function other responses to such a request are possible, such as presenting an alert or quitting the program altogether.

The `Mouse` function defines the response of the window to mouse events in the window's contents. The `MouseState` contains information about the mouse event: the position of the pointer (in picture domain co-ordinates), whether it was a click, a double-click, a triple-click or a release of the mouse button and which modifier keys (shift, option etc.) were being held down.

The `Keyboard` function is very similar to the mouse function. It defines the window's response to keyboard events. All keyboard events are directed to the active window. The `KeyboardState` contains all information needed to handle the event: which key was pressed or released and which of the modifier keys were being held down at that time. The library module `deltaSystem` contains macro's for a number of special keys such as the arrow keys.

With the `Cursor` attribute the shape of the mouse pointer inside the contents of the window can be defined. Several `CursorShapes` have been predefined (arrow, I-beam, cross etc.). When no cursor definition is given the cursor will have the global cursor shape inside the window's contents.

When the `StandByWindow` constructor occurs in the attribute list the window becomes a so-called stand-by window. On some operating systems a window must be activated explicitly by clicking on it. The mouse events related to that click are usually not passed to the mouse event handler of that window. A stand-by window also accepts these mouse events: when the window is activated by the user by clicking somewhere in the contents of the window the mouse event handler of the window will be called immediately after the window has been activated.

Drawing in windows

The library module `deltaWindow` contains several functions with which one can draw in a window:

```
:: DrawInWindow WindowId [DrawFunction] (IOState s) -> IOState s;
:: DrawInActiveWindow    [DrawFunction] (IOState s) -> IOState s;
```

These functions take a list of `DrawFunctions` and apply these to the indicated window in order of appearance in the list. A large set of functions has been predefined in the module `deltaPicture`. Font handling (useful when texts are drawn in a window) is performed by functions from the library module `deltaFont`. As said earlier a `DrawFunction` is a function of type => `Picture Picture`. A `Picture` can be seen as the environment through which drawing takes place. Each window has its own `Picture` environment. The following example is a function that draws a string in a window at a certain position:

```
:: DrawStringInWindow WindowId (INT,INT) STRING (IOState s)
                                                     -> IOState s;
   DrawStringInWindow id pos string io
   -> DrawInWindow id [MovePenTo pos, DrawString string] io;
```

`MovePenTo` and `DrawString` are predefined functions from the deltaPicture library module.

The following predefined functions make it possible to explicitly apply an `UpdateFunction` to the picture of a window:

```
:: DrawInWindowFrame WindowId (UpdateFunction s) s (IOState s)
                                                -> (s, IOState s);
:: DrawInActiveWindowFrame    (UpdateFunction s) s (IOState s)
                                                -> (s, IOState s);
```

These functions are useful when one wants to redraw the complete picture. The `UpdateArea` passed to the `UpdateFunction` is the visible part of the window's picture, which makes it possible to redraw the visible part only (very useful when redrawing the complete picture is time consuming). Apart from that it is also possible to retrieve the visible part of a window's picture explicitly.

Changing windows during the interaction

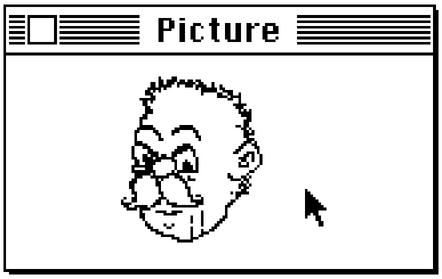
The module `deltaWindow` also contains various functions to change parts of the definition of a window during the interaction. In fact all attributes of a window can be changed, except of course the id and the position and size, which are in control of the user. Functions have been predefined to change the title, the settings of the scroll bars, the picture domain, and all event handlers: the update, activate, deactivate, go-away, mouse and keyboard function. The mouse and the keyboard can also be disabled and enabled. Furthermore the local cursor shape of the window can be changed. The module `deltaIOState` contains rules that affect the global cursor, i.e. the cursor shape outside windows with a local cursor shape definition.

An example of a window-based interaction

The following example is a simple drawing program. It defines one fixed size window in which pixels can be drawn with the mouse:

```
MODULE SimpleDraw;

IMPORT deltaEventIO, deltaWindow, deltaPicture;

TYPE
::  UNQ S  -> [Point];
::  UNQ IO -> IOState S;

RULE
::  Start UNQ WORLD -> UNQ WORLD;
    Start world -> CloseEvents events' world',
        (s,events'): StartIO [WindowSystem [window]] [] [] events,
```

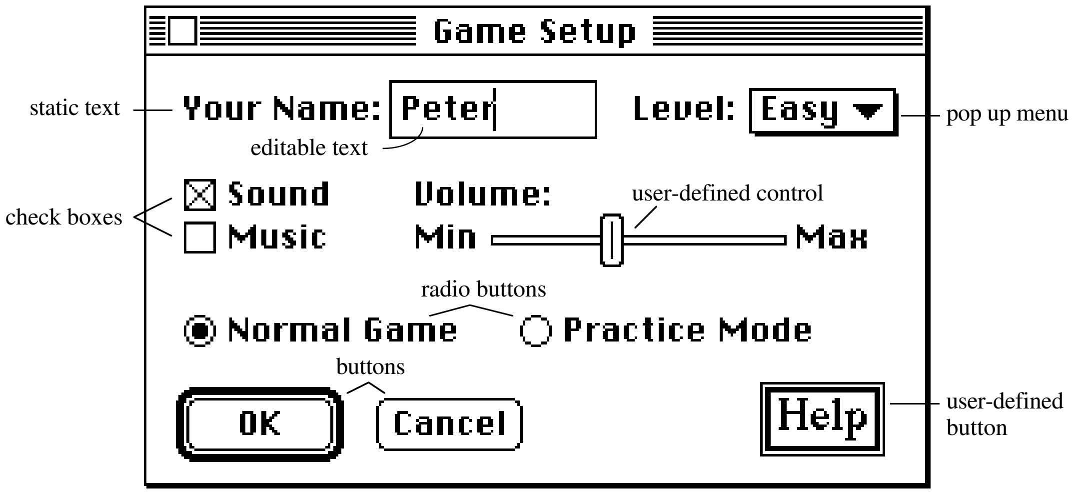
```
        (events,world'): OpenEvents world,
        window: FixedWindow 1 (0,0) "Picture" ((0,0),(160,80))
                    UpdatePicture [Mouse Able Track, GoAway Quit];

:: UpdatePicture UpdateArea S -> (S, [DrawFunction]);
   UpdatePicture area s:[hd|tl] -> (s, Map DrawPoint [hd|tl]);
   UpdatePicture area [] -> ([],[]);

:: Track MouseState S IO -> (S, IO);
   Track (pos, ButtonUp  , mods) s io -> (s, io);
   Track (pos, buttondown, mods) s io
   -> ([pos|s], DrawInActiveWindow [DrawPoint pos] io);

:: Quit S IO -> IO;
   Quit s io -> (s, QuitIO io);

:: Map (=> t s) [t] -> [s];
   Map f [h|t] -> [f h | Map f t];
   Map f [] -> [];
```



A simple but complete drawing program and the way it looks on a Macintosh system. The mouse function does nothing when the mouse button is released. When the button is down the position is added to the program state (to be able to update the window) and the point is drawn in the window. The update function just maps the predefined function `DrawPoint` over the list of points in the program state. This program is not very efficient: even when the mouse does not move points are drawn and added to the program state. It is easy to modify the `Track` function to check first whether the mouse has actually moved.

5.6 Dialogs and notices

A dialog is a kind of window in which communication between the user and the program is structured in some way. It is generally used when the application needs more information from the user to carry out a command. It resembles a form in which the user checks boxes and fills in blanks. A dialog can be modeless or modal. A modal dialog forces the user to deal with the dialog entirely before any other action can take place. Modeless dialogs are less demanding: even when they are open the user may disregard them and use them whenever that is convenient. Just like with windows only one dialog is active at a time. All keyboard events are directed to the active dialog. One of the buttons in a dialog, if any, is called the default button. It can not only be selected by the mouse, but also by pressing return or enter. The default button is usually recognizable by a bold outline. The following picture shows a typical dialog and clarifies some of the terminology used in this section.



A (modeless) dialog.

In Concurrent Clean the predefined algebraic type `DialogDef` is used to specify dialogs such as the one in the picture:

```
:: DialogDef UNQ s UNQ io
   -> PropertyDialog    DialogId DialogTitle [DialogAttribute]
                        (SetFunction s io) (ResetFunction s io)
                        [DialogItem s io]
   -> CommandDialog     DialogId DialogTitle [DialogAttribute]
                        DialogItemId [DialogItem s io];
:: DialogId          -> INT;
:: DialogTitle       -> STRING;
:: DialogAttribute   -> DialogPos    Measure Measure
                     -> DialogSize   Measure Measure
                     -> DialogMargin Measure Measure
                     -> ItemSpace    Measure Measure
                     -> StandByDialog;
:: Measure           -> MM REAL | Inch REAL | Pixel INT;
:: SetFunction    UNQ s UNQ io -> ButtonFunction s io;
:: ResetFunction  UNQ s UNQ io -> ButtonFunction s io;
:: ButtonFunction UNQ s UNQ io
   -> => DialogInfo (=> s (=> io (s,io)));
```

A difference is made between property dialogs and command dialogs. A property dialog is a modeless dialog that is used to set certain values or properties (default values, preferences etc.) of an interaction. It has two predefined buttons: The set and the reset button. With the set button the values entered in the dialog should become valid, whereas pressing the reset button should result in resetting the values in the dialog to certain default values. The set button is always the default button of a property dialog. A command dialog can be modal as well as modeless and is used to carry out a command which requires certain parameters that cannot be set via the menu system. Dialogs defined in the initial I/O system will be opened at the beginning of the interaction as modeless dialogs. During the interaction modal as well as modeless dialogs can be opened and closed by means of functions from the `deltaDialog` library module, as will be explained later on. A `PropertyDialog` is defined by the following parameters:

- `DialogId`: An integer uniquely identifying this dialog. When it is tried to open a dialog with the same id as a dialog that is already open, the dialog will not be opened. Instead the existing dialog with the matching id will be activated.

- `DialogTitle`: The title of the dialog.

- A list of `DialogAttributes`: With these attributes the exact position and size of the dialog can be defined. Furthermore the distance between the dialog items and the borders of the dialog (margins) and the distance between the items of the dialog (item spaces) can be defined. The first parameter of each of the attributes always represents the horizontal attribute value, the second always represents the vertical attribute value. When none of these items is specified the dialog is centred on the screen, a size is chosen such that all items fit in the dialog and save default margins and item spaces are chosen. The attribute values can be specified in millimetres, inches or screen pixels. The `deltaSystem` library module contains functions to explicitly convert millimeters and inches into screen pixels. When the `StandByDialog` constructor occurs in the attribute list the dialog becomes a so-called stand-by dialog. A stand-by dialog behaves exactly like a stand-by window.

- `SetFunction` / `ResetFunction`: The event handlers for the predefined set and reset buttons. When one of these buttons is pressed the corresponding function will be called. These functions get the current `DialogInfo` of the dialog as parameter, which is an abstract type representing the current settings of the dialog items. The library module `deltaDialog` contains functions to retrieve information such as the contents of editable text fields and the settings of check boxes from the `DialogInfo`.

- A list of `DialogItems`: The definition of the other items in the dialog, such as text fields, check boxes, pop up menus etc. In the following section the definition of dialog items is treated.

A `CommandDialog` also has an id, a title, a list of attributes and a list of dialog items as parameters. Besides these, it has the id of its default button as parameter. When this id is not the id of a button, the first button in the list of dialog items is chosen to be the default button.

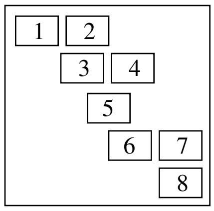
Dialog items

In the picture of a dialog presented earlier a lot of different kinds of dialog items can be distinguished. For each of these kinds the predefined algebraic type `DialogItem` contains an alternative:

```
:: DialogItem UNQ s UNQ io
   -> DialogButton DialogItemId ItemPos ItemTitle SelectState
                   (ButtonFunction s io)
   -> DialogIconButton DialogItemId ItemPos PictureDomain
                   IconLook SelectState (ButtonFunction s io)
   -> StaticText DialogItemId ItemPos STRING
   -> DynamicText DialogItemId ItemPos TextWidth STRING
   -> EditText DialogItemId ItemPos TextWidth NrEditLines STRING
   -> DialogPopUp DialogItemId ItemPos SelectState DialogItemId
                   [RadioItemDef s io]
   -> RadioButtons DialogItemId ItemPos RowsOrColumns
                   DialogItemId [RadioItemDef s io]
   -> CheckBoxes DialogItemId ItemPos RowsOrColumns
                   [CheckBoxDef s io];
   -> Control DialogItemId ItemPos PictureDomain SelectState
                   ControlState ControlLook ControlFeel
                   (DialogFunction s io);
:: RadioItemDef UNQ s UNQ io
   -> RadioItem DialogItemId ItemTitle SelectState
                   (DialogFunction s io);
:: CheckBoxDef UNQ s UNQ io
   -> CheckBox DialogItemId ItemTitle SelectState MarkState
                   (DialogFunction s io);
:: DialogItemId  -> INT;
:: ItemPos       -> Left | Center | Right |
                    RightTo DialogItemId | Below DialogItemId |
                    XOffset DialogItemId Measure |
                    YOffset DialogItemId Measure |
                    XY Measure Measure | ItemBox INT INT INT INT;
:: IconLook      -> => SelectState [DrawFunction];
:: TextWidth     -> Measure;
:: NrEditLines   -> INT;
:: RowsOrColumns -> Rows INT | Columns INT;
:: DialogFunction UNQ s UNQ io -> => DialogInfo
                  (=> (DialogState s io) (DialogState s io));
```

All dialog items have an id and a position (`ItemPos`). When this position is `Left`, `Center` or `Right` the item is placed left-aligned, right-aligned or centred, beneath all items defined earlier in the list. When this position is `RightTo`, `Below`, `XOffset` or `YOffset` the item is placed relative to the item with the indicated id. When that id is not the id of an item that is defined earlier in the list of items, the item is placed beneath the items defined earlier, left-aligned. When an item is placed below an item that is centred resp. right-aligned this item will also be centred resp. right-aligned. When an item is placed right to a centred item, these items are centred together. With the `XY` constructor the exact horizontal and vertical co-ordinates of the item can be defined (the top-left corner of the dialog is (0,0)). With the `ItemBox` constructor the horizontal and vertical position and the width and height of the item can be defined in terms of screen pixels. The `XY` and `ItemBox` constructor should not be used when you want your program to be portable. It is clear that, because items can only be placed below or right to items defined earlier in the list, the first item in the list must be the top-left item. An example of the use of the item positions:

```
[Item 1 Left,
 Item 2 (RightTo 1),
 Item 3 Center,
 Item 4 (RightTo 3),
 Item 5 (Below 3), (or: Below 4)
 Item 6 Right,
 Item 7 (RightTo 6),
 Item 8 (Below 6)] (or: Below 7)
```



The resulting dialog layout

A `DialogButton` is usually pressed by the user when she/he is done with the dialog. A button has a title, which will be shown inside the button, and a `ButtonFunction`: the event handler that will be called when the button is pressed. As said earlier such an event handler gets the current settings of the dialog as parameter. How a `DialogButton` looks is system dependent. A `DialogIconButton` is a button with a program-defined look (i.e. system independent). Instead of a title it has a `PictureDomain` as parameter and a function defining this look by means of a list of `DrawFunctions`. A `DialogIconButton` that is not selectable (`Unable`) actually just defines a picture in the dialog. A dialog is not automatically closed when a button is pressed. When the dialog should be closed when a button is pressed this must be specified explicitly in the event handler of the button.

A `StaticText` is a piece of text that is shown in the dialog. It can neither be changed by the program nor edited by the user. A `DynamicText` is a piece of text that can be changed by the program during the interaction. The `TextWidth` parameter defines how many room will be reserved for the text in the dialog. An `EditText` is an editable text field. This text can be changed by both the user and the program. Both the width of the text field and the number of lines that will fit inside the field must be defined. The text inside the field can be edited using the keyboard and the mouse.

A `DialogPopUp` defines a pop up menu that contains radio items. A group of radio items represent a number of options of which only one is valid at a time. The menu appears when the user clicks on the pop up box: a rectangle that shows the currently selected option. The second `DialogItemId` parameter of the `DialogPopUp` constructor is the id of the radio item that is selected initially. The items are defined by a list of `RadioItemDefs`, which have a special kind of event handler as parameter: a so-called `DialogFunction`. Instead of the program state and `IOState` a `DialogFunction` has the current settings of the dialog (`DialogInfo`) and a `DialogState` as parameters. It returns a new `DialogState`. `DialogState` is an `UNQ` abstract type that represents the state of the dialog. Therefore a `DialogFunction` can only change the settings of the dialog, but it cannot affect the rest of the interaction. In the module `deltaDialog` a number of functions of type `=> DialogState DialogState` has been predefined. A `DialogFunction` can for example be used to disable or enable items in a dialog, to change the contents of a `DynamicText` etc.

The `RadioButtons` constructor also defines a group of radio items. The difference with a `DialogPopUp` is that all items of the group are visible in the dialog. Therefore radio buttons are usually used when the number of radio items is small. When placing the items in the dialog a group of radio buttons is considered to be one item. The place of the radio buttons inside the group is defined by the `RowsOrColumns` parameter, which defines the number of rows or columns in which the radio buttons should appear.

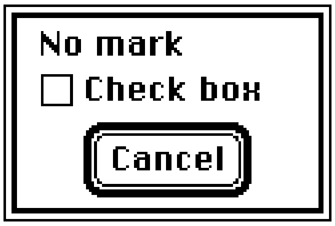
The `CheckBoxes` constructor defines a group of check boxes. With a check box an option can be set on or off. Clicking on a check box will revert its setting. Just like radio buttons a group of check boxes is considered to be one item, whereas the place of the check boxes inside the group is defined by the `RowsOrColumns` parameter.

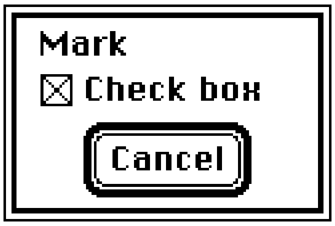
The following example shows the definition of a concrete dialog. It is a modal dialog that contains one check box. The setting of the check box determines which text will be shown in the dialog (Mark or No mark). The Cancel button just closes the dialog:

```
TYPE
::   UNQ S -> ...;    :: UNQ IO -> IOState S;
RULE
::   MyDialog -> DialogDef S IO;
     MyDialog -> CommandDialog 1 "Example" [] 3
          [DynamicText 1 Left (MM 25.0) "No mark",
```

```
            CheckBoxes 2 Left (Rows 1)
                [CheckBox 21 "Check box" Able NoMark ChangeText],
            DialogButton 3 Center "Cancel" Able Cancel];

::  ChangeText DialogInfo (DialogState S IO)
                                -> DialogState S IO;
    ChangeText dinfo dstate
    -> ChangeDynamicText 1 "Mark"    dstate, IF marked
    -> ChangeDynamicText 1 "No mark" dstate,
         marked: CheckBoxMarked 21 dinfo;

::  Cancel DialogInfo S IO -> (S, IO);
    Cancel dinfo s io -> (s, CloseActiveDialog io);
```

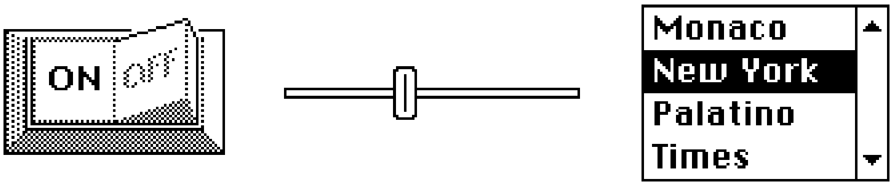
`ChangeDynamicText`, `CheckBoxMarked` and `CloseActiveDialog` are predefined functions from the `deltaDialog` module. Depending on the setting of the check box the dialog will look as follows (when opened as a modal dialog on an Apple Macintosh system):





Controls

Although all frequently used dialog items have been predefined in Concurrent Clean, it is also possible to define your own dialog items. These user-defined items, so-called *controls*, can be used when one needs a very sophisticated or specialised dialog item. Contrary to the predefined dialog items, these controls will have a look and feel that is independent of the graphical user interface system that is used: The look and feel of the control is completely defined by the program.



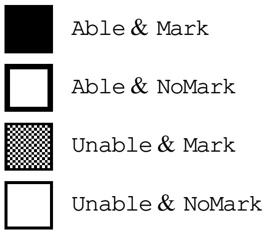
Examples of program-defined controls: a switch, a slider bar and a scrolling list.

In Concurrent Clean a control is just a `DialogItem`:

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

:: ControlState  -> BoolCS BOOL | IntCS INT | RealCS REAL |
                    StringCS STRING | ListCS [ControlState] |
                    PairCS ControlState ControlState;
:: ControlLook   -> => SelectState (=> ControlState
                         [DrawFunction]);
:: ControlFeel   -> => MouseState (=> ControlState
                         (ControlState, [DrawFunction]));
```

A `Control` intuitively defines a kind of fixed window inside the dialog, with a picture domain, an update function (`ControlLook`) and a mouse function (`ControlFeel`). The `ControlState` defines the state of the control. When for example a switch is defined this state will probably be a boolean indicating the setting of the switch. The `ControlLook` gets the `SelectState` and of course the `ControlState` of the control as arguments and returns a list of draw functions that are drawn in the picture domain of the control. It is guaranteed by the system that no drawing will take place outside this domain. The `ControlFeel` is called whenever a mouse event occurs in the control's picture domain. It gets the `MouseState` and the current `ControlState` as arguments. It returns a new `ControlState` and a list of draw functions that reflect the effect of the mouse event on the control's picture. It is important to note that before draw functions are applied to a picture in a dialog, the font is always set to the default font, the pen to normal and the pen colour to black. The following example defines an alternative for the predefined check boxes:

```
MACRO MarkBox -> ((0,0),(18,18));
RULE
:: MyCheckBox DialogItemId ItemPos SelectState MarkState
                    (DialogFunction s io) -> DialogItem S IO;
   MyCheckBox id pos ability mark function
   -> Control id pos MarkBox ability (BoolCS (Marked mark))
              Look Feel function;

:: Look SelectState ControlState -> [DrawFunction];
   Look ability (BoolCS marked)
   -> [fill]                            , IF able && marked
   -> [SetPenSize (2,2), draw]          , IF able
   -> [SetPenPattern GreyPattern, fill,
        SetPenPattern BlackPattern, draw], IF marked
   -> [draw],
      draw: DrawRectangle MarkBox,
      fill: FillRectangle MarkBox,
      able: Enabled ability;

:: Feel MouseState ControlState -> (ControlState,[DrawFunction]);
   Feel (pos,ButtonUp,mods) (BoolCS marked)
   -> (state', [EraseRectangle MarkBox | Look Able state']),
      state': BoolCS (NOT marked);
   Feel mstate cstate -> (cstate, []);

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

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



A variation on check boxes implemented as a `Control`. The look of the check box is defined by `Look`. There are four variations of the check box, depending on the `SelectState` and the `MarkState` of the control. The behaviour of the check box is defined by `Feel`. When the mouse button is released (`ButtonUp`) the state of the control is reversed and the check box is redrawn by calling the `Look` with the new state.

The library module `deltaDialog` contains functions to disable and enable controls and to change the state, look and feel of a control. When the look is changed (by means of the predefined function `ChangeControlLook`), the control is completely redrawn. Furthermore the current `ControlState` of a `Control` can be retrieved from the `DialogInfo` parameter of `ButtonFunctions` and `DialogFunctions`.

Two useful controls have been predefined in the event I/O library (module `deltaControls`): slider bars and scrolling lists. Slider bars can be horizontal or vertical. The length of a slider bar is defined by the program. A scrolling list contains a number of strings, of which one is selected. With a kind of scroll bar it is possible to scroll through the list. The `deltaControls` module contains functions to define and change the predefined controls and to retrieve their current settings (see appendix C). The picture presented earlier in this section shows a slider bar and a scrolling list.

Opening and changing dialogs during the interaction

Modal as well as modeless dialogs can be opened during the interaction. The following function opens a modeless (property or command) dialog:

```
:: OpenDialog (DialogDef s (IOState s)) (IOState s) -> IOState s;
```

It takes the definition of a property or command dialog and the current `IOState` as arguments, opens the corresponding modeless dialog, terminates and returns the new `IOState`. It is also possible to open command dialogs as modal dialogs:

```
:: OpenModalDialog (DialogDef s (IOState s)) s (IOState s)
   -> (s, IOState s);
```

This function opens the modal dialog and then takes over all event handling until the dialog is closed (by means of a call to `CloseDialog` or `CloseActiveDialog`). When the dialog is closed the function terminates with the resulting program state and `IOState`. It is possible to open a modal dialog when there is already one active. The modal dialog that is opened most recently will be the one to which all events are directed. Modeless dialogs and windows that are opened when a modal dialog is active will be opened underneath the active modal dialog: the dialog will remain active. Attempts to open `PropertyDialogs` with `OpenModalDialog` are ignored. Both functions do not open dialogs with the same id as a dialog that is already open. When this is tried the open dialog with the same id is activated instead.

As explained earlier local changes to a dialog can be made through the `DialogFunctions` of the items in the dialog. But, it is also possible to modify open dialogs from outside the dialog. The following function has been predefined in the `deltaDialog` library module:

```
:: ChangeDialog DialogId [DialogChange s] (IOState s)
                                                     -> IOState s;
```

A `DialogChange` is a function of type => `(DialogState s (IOState s)) (DialogState s (IOState s))`, which is the type of most functions in the `deltaDialog` module. The `ChangeDialog` function applies the list of `DialogChanges` to the dialog with the specified id in order of appearance in the list. When the dialog with the specified id cannot be found the changes are ignored.

The `deltaDialog` library contains functions to disable and enable dialog items, to mark and unmark check boxes, to explicitly select a radio item, to change the contents of dynamic and editable text fields, to change the look of program defined buttons and controls, to change state and feel of controls and to change the event handlers related to dialog items (button functions and dialog functions). These functions can be used as `DialogFunctions` as well as in the `ChangeDialog` function.

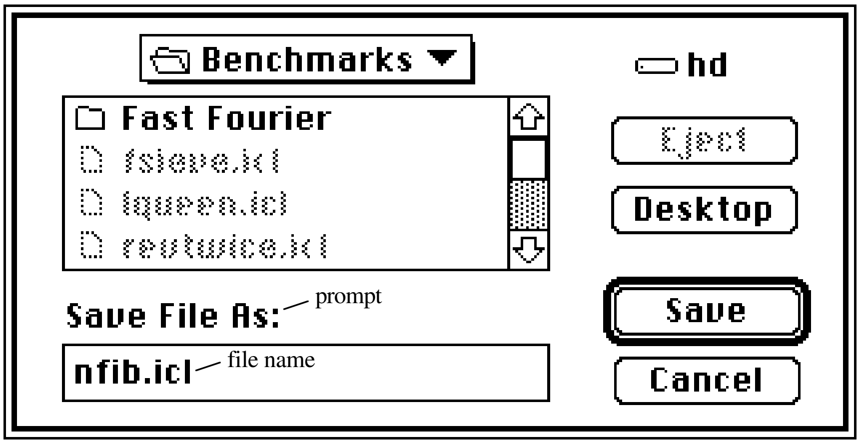
The file selector dialogs

For the selection of files from the file system two dialogs have been predefined in the library module `deltaFileSelect`. These predefined dialogs can be opened by the following functions:

```
:: SelectInputFile FILES (IOState s)
                     -> (BOOL, STRING, FILES, IOState s);
:: SelectOutputFile STRING STRING FILES (IOState s)
                     -> (BOOL, STRING, FILES, IOState s);
```

The `SelectInputFile` function presents a dialog to select an existing file. It returns boolean indicating whether the Open button (`TRUE`) or the Cancel button (`FALSE`) was pressed and a string containing the pathname of the selected file. When the Cancel button was pressed an empty string will be returned. The pathname can be used as parameter for the functions `FOpen` and `SFOpen` of the `deltaFile` module to open the file.

`SelectOutputFile` has two strings as parameter. The first string is the text that will be presented in the dialog (the prompt). When this string is empty the default prompt ("Save As:") will be chosen. The second parameter is the default name for the file to be selected. This default name can be changed in the dialog. Just like `SelectInputFile`, the `SelectOutputFile` function returns a boolean indicating whether the Save button (`TRUE`) or the Cancel button (`FALSE`) was pressed and a string containing the pathname of the selected file. When the Cancel button was pressed an empty string will be returned. When the selected file already existed in the indicated directory an alert will appear, asking whether the selected file may be overwritten.



Example of a dialog to select an output file. The layout of this dialog depends on the interface system, but this example gives a good idea about how such dialogs usually look.

The about-dialog

The about-dialog is a very special dialog. It should be used to show information about the application (version, authors etc.). It may also contain a button that should provide a help facility. The about-dialog of an application will be accessible by the user in a system dependent way. On a Macintosh for example the about-dialog can be opened with a special menu command in a system defined menu (the 'About...' command of the Apple () menu). In Concurrent Clean the about-dialog can be defined in the initial dialog device:

```
:: DialogDef UNQ s UNQ io
   -> ...
   -> AboutDialog ApplicationName PictureDomain [DrawFunction]
                  (AboutHelpDef s io);
:: ApplicationName -> STRING;
:: AboutHelpDef UNQ s UNQ io
   -> AboutHelp ItemTitle (AboutHelpFunction s io)
   -> NoHelp;
:: AboutHelpFunction UNQ s UNQ io -> => s (=> io (s,io));
```

The first `AboutDialog` that is encountered in the initial dialog device becomes the about-dialog of the application. Attempts to open `AboutDialogs` with `Open(Modal)Dialog` are ignored. It is not possible to change the about-dialog during the interaction. When no about-dialog is defined a default about-dialog will be provided by the library.

The type definition above shows that an about-dialog is just a picture (defined by a picture domain and a list of draw functions) together with an optional button (`AboutHelpDef`). This button has a title and an event handler, which is called when the button is pressed. This event handler (`AboutHelpFunction`) should provide a help facility but may have any effect.

Notices

A notice is used to inform the user about unusual or possibly dangerous situations. The simplest kind of notice is a beep, which is provided by the following function from the `deltaDialog` module:

```
:: Beep (IOState s) -> IOState s;
```

Usually one wants to give the user more information about the situation. For this purpose the event I/O library provides notices: simple, modal dialogs containing only text and final buttons:

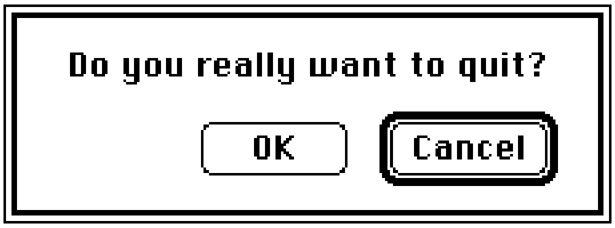
```
:: NoticeDef       -> Notice [STRING] NoticeButtonDef
                             [NoticeButtonDef];
:: NoticeButtonDef -> NoticeButton NoticeButtonId ItemTitle;
:: NoticeButtonId  -> INT;
```

The list of strings defines the message of the notice: each string is a line of this message. The first `NoticeButton` is the default button of the notice, the other buttons are defined by the list of notice buttons. A `NoticeButton` only has an id and a title. The notice is closed automatically when one of the buttons is pressed. A notice can be opened with the following function from the `deltaDialog` library module:

```
:: OpenNotice NoticeDef (IOState s)
                   -> (NoticeButtonId, IOState s);
```

This function opens the notice and terminates when the user presses one of the buttons of the notice. It returns the id of the button that was pressed. Example of the use of a notice:

```
:: Quit s (IOState s) -> (s, IOState s);
   Quit s io -> (s, io'), IF = id 1
             -> (s, QuitIO io'),
       (id,io'): OpenNotice notice io,
       notice  : Notice ["Do you really want to quit?"]
                        (NoticeButton 1 "Cancel")
                        [NoticeButton 2 "OK"];
```



A `Quit` function that asks the user to confirm before quitting. The picture shows the notice that will be presented (on a Macintosh system).

5.7 The timer device

The timer device contains so-called timers. A timer is used when one wants to perform some continuous action each time a certain time interval has passed. A timer is defined by the following algebraic data type:

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

Like many other interface objects a timer has an id and a `SelectState` (`Able` or `Unable`). The `TimerInterval` is defined as a number of ticks. The number of ticks per second depends on the operating system. A macro `TicksPerSecond` is available in the library module `deltaTimer`. The `TimerFunction` is the event handler that is called each time the timer interval has passed. The `TimerState` parameter contains the number of times the interval has passed since the last time the timer function was called. This number might be greater than one: the program might be busy evaluating another event handler (each new event is handled only after the current event handler has returned a complete `IOState`). The evaluation of this event handler might take more time than one timer interval. In that case the `TimerState` will be greater than one. It is guaranteed however, that each timer gets its turn some time, provided that no non-terminating event handlers have been defined. When the `TimerInterval` of a timer is smaller than one, the timer function of this timer will be called as often as possible. The `TimerState` argument of the timer function will then always be one. The following example is an irritating program that will beep once per second for one minute:

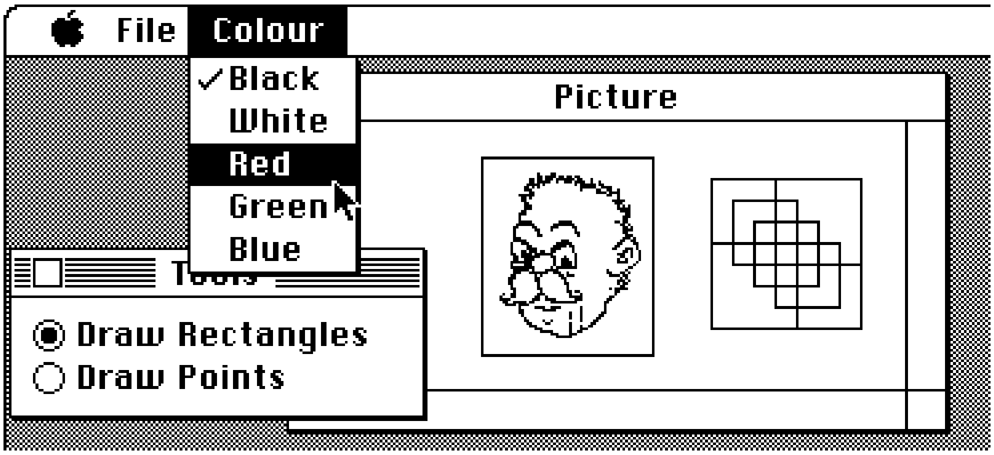
```
MODULE Timers;

IMPORT deltaEventIO, deltaIOSystem, deltaTimer;

TYPE
::   UNQ S -> INT;      ::    UNQ IO -> IOState S;

RULE
::   Start UNQ WORLD -> UNQ WORLD;
     Start w -> CloseEvents e' w',
          (s,e'): StartIO [TimerSystem [beeps, quit]] 0 [] e,
          (e,w'): OpenEvents w,
          beeps : Timer 1 Able TicksPerSecond        Beeps,
          quit  : Timer 2 Able (* TicksPerSecond 60) Quit;

::   Beeps TimerState S IO -> (S, IO);
     Beeps tstate s io -> (s, Beep io);

::   Quit TimerState S IO -> (S, IO);
     Quit tstate s io -> (s, QuitIO io);
```

Two timers are defined in this program. The first timer is responsible for the beeps that are emitted once per second, the second one just quits the interaction (and the program) after one minute.

The library module `deltaTimer` contains functions to add and delete timers during the interaction, to enable and disable timers, and to change the timer function and interval of timers. Furthermore it contains functions that delay the evaluation of an expression for a certain time interval. Last but not least it contains functions that return the current time and date.

5.8 A drawing program

In this section we present an example of a complete interactive Concurrent Clean program. It is a simple drawing program in which pictures consisting of points and rectangles in five different colours can be drawn with the mouse.



The drawing program on the screen of an Apple Macintosh.

The program defines two menus. One is called File and contains the Quit command and the command that opens or activates the Tools dialog. The other menu is called Colour and is used to select the colour in which drawing will take place. The Tools dialog is the only dialog of the program. It is opened at the beginning of the interaction and contains two radio buttons. The settings of the radio buttons indicate which figure will be drawn (rectangles or points). To keep the program simple there is only one window in which drawing takes place. This window is opened at the beginning of the interaction.

```
MODULE TinyDraw;

IMPORT deltaI, deltaB;
IMPORT deltaEventIO, deltaIOSystem, deltaPicture, deltaWindow, deltaDialog;
```

The type block defines the program state State and a shorthand for IOState to prevent type rules from cluttering up. The program state consists of the position of the mouse, the colour, the figure that is being drawn and the figures that have been drawn earlier. A figure is either a rectangle or a list of points with a certain colour.

```
TYPE
::  UNQ IO       -> IOState State;
::  UNQ State    -> ( Point, Colour, Figure, [Figure]);
::  Figure       -> RectF    Colour Rectangle
                 -> PointsF  Colour [ Point];

RULE
```

The execution of the program starts here. The event stream is retrieved from the world and passed to StartIO. The I/O system consists of the menus File and Colour, the Picture window and the Tools dialog. The initial program state reflects the fact that the initial colour is black and that no figures have been drawn yet. The resulting program state of the interaction is ignored and the resulting event stream is put back into the world.

```
::  Start UNQ WORLD        ->  UNQ WORLD;
    Start world           ->  CloseEvents events' world',
        (s, events')      :   StartIO [menu, window, dialog] initstate [] events,
        (events, world')  :   OpenEvents world,
        initstate         :   ((0,0), BlackColour, PointsF BlackColour [], []),

        menu  : MenuSystem [file, color],
        file  : PullDownMenu FileId "File" Able [
                    MenuItem  ToolsId  "Tools..."  (Key 'T')  Able  ShowTools,
                    MenuSeparator,
                    MenuItem  QuitId   "Quit"      (Key 'Q')  Able  Quit],
        color : PullDownMenu ColorId "Colour" Able [
                    MenuRadioItems BlackId [
                        MenuRadioItem  BlackId  "Black"  NoKey  Able  (SetColour BlackColour),
                        MenuRadioItem  WhiteId  "White"  NoKey  Able  (SetColour WhiteColour),
                        MenuRadioItem  RedId    "Red"    NoKey  Able  (SetColour RedColour),
                        MenuRadioItem  GreenId  "Green"  NoKey  Able  (SetColour GreenColour),
                        MenuRadioItem  BlueId   "Blue"   NoKey  Able  (SetColour BlueColour)]],

        window: WindowSystem [ ScrollWindow 1 (0,0) "Picture"
                        (ScrollBar (Thumb 0) (Scroll 10))  (ScrollBar (Thumb 0) (Scroll 10))
                        ((0,0), (1000,1000)) (50,50) (500,300) Update [Mouse Able StartMouse,
                                        GoAway Quit, Cursor CrossCursor, StandByWindow]],

        dialog: DialogSystem [ToolsDialog];
```

The event handler for the Show Tools... command.

```
::  ShowTools State IO -> (State, IO);
    ShowTools state io -> (state, OpenDialog ToolsDialog io);
```

The event handler for the Quit command.

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

The event handler for the colours in the Colour menu. It changes the colour in the program state and sets the chosen colour in the picture of the window. To reduce the size of the program this function is used curried in its first argument in the Start rule. This way one event handler can be used for five menu items.

```
:: SetColour Colour State IO -> (State, IO);
   SetColour color (pos,oldc,fig,figs) io
   -> ((pos,color,fig,figs), DrawInActiveWindow [SetPenColour color] io);
```

The definition of the Tools dialog.

```
:: ToolsDialog -> DialogDef State IO;
   ToolsDialog -> CommandDialog ToolsDialogId "Tools"
                    [StandByDialog, DialogMargin (Pixel 6) (Pixel 6)] 0 [
                       RadioButtons FiguresId Left (Rows 2) RectItemId [
                          RadioItem RectItemId    "Draw Rectangles"  Able  SelectMode,
                          RadioItem PointsItemId  "Draw Points"      Able  SelectMode ]];
```

The DialogFunction of the radio buttons. This function does nothing.

```
:: SelectMode DialogInfo (DialogState State IO) -> DialogState State IO;
   SelectMode dinfo dstate -> dstate;
```

The initial mouse function of the Picture window. When the mouse button is pressed it is tried to find out which of the figures should be drawn. When the dialog is open the settings of the radio buttons define which figure should be drawn (NewCurrentFigure). When it has been closed the previously drawn figure defines which figure should be drawn (OldCurrentFigure). The mouse function is changed to Track, which does the real drawing.

```
:: StartMouse MouseState State IO -> (State, IO);
   StartMouse (pos, ButtonDown, modifiers) state io
   -> (state1, DrawInActiveWindow [SetPenMode mode1] io'), IF open
   -> (state2, DrawInActiveWindow [SetPenMode mode2] io'),
      (mode1,state1)   : NewCurrentFigure pos state (GetSelectedRadioItemId FiguresId dinfo),
      (mode2,state2)   : OldCurrentFigure pos state,
      (open,dinfo,io') : GetActiveDialogInfo (ChangeActiveMouseFunction Track io);
   StartMouse mouse state io -> (state, io);
```

Depending on the settings of the radio buttons NewCurrentFigure selects a new figure. It returns a program state containing a new empty current figure and a pen mode that is suitable for the figure to be drawn. OldCurrentFigure does the same, but uses the previously drawn figure to select the new current figure.

```
:: NewCurrentFigure Point State DialogItemId -> (PenMode, State);
   NewCurrentFigure pos (oldpos,color,fig,figs) figureid
   -> (XorMode,   (pos, color, RectF color (pos, pos), figs)), IF = figureid RectItemId
   -> (CopyMode,  (pos, color, PointsF color [pos], figs));

:: OldCurrentFigure Point State -> (PenMode, State);
   OldCurrentFigure pos (oldpos,color,RectF col ps,figs)
   -> (XorMode,   (pos, color, RectF color (pos, pos), figs));
   OldCurrentFigure pos (oldpos,color,PointsF col ps,figs)
   -> (CopyMode,  (pos, color, PointsF color [pos], figs));
```

Track is the mouse function that does the real drawing. When the button is released the figure has been completed. It is added to the list of figures and the mouse function is changed back to StartMouse. When the button is down it is first checked whether the mouse has moved. If not, nothing is done. Otherwise the figure is changed according to the mouse position and the effect of this change is shown in the window.

```
:: Track MouseState State IO -> (State, IO);
   Track (pos, ButtonUp, mods) (oldpos, color, fig, figs) io
   -> ((pos, color, fig, [fig | figs]), ChangeIOState [mouse, draw] io),
      mouse   : ChangeActiveMouseFunction StartMouse,
      draw    : DrawInActiveWindow (FinishFigure fig);
   Track (pos, buttondown, mods) state:(oldpos, color, fig, figs) io
   -> (state, io), IF EqPoint pos oldpos
   -> ((pos, color, fig', figs), DrawInActiveWindow drawfs io),
      (fig', drawfs): ChangeFigure pos fig;
```

ChangeFigure returns the changed figure and a list of draw functions that reflect this change according to the new position of the mouse. When the current figure is a rectangle the old rectangle is erased and the new rectangle is drawn (rectangles are drawn in XOR mode!). When points are drawn the current mouse position is added to the list of points representing the figure and the point is drawn.

```
:: ChangeFigure Point Figure -> (Figure, [DrawFunction]);
   ChangeFigure newpos (RectF color oldrect:(startpos, oldpos))
   -> (RectF color newrect, [DrawRectangle oldrect, DrawRectangle newrect]),
      newrect: (startpos, newpos);
   ChangeFigure newpos (PointsF color points)
   -> (PointsF color [newpos | points], [DrawPoint newpos]);
```

During tracking rectangles are drawn in XOR mode. When the user has finished a rectangle the rectangle must be redrawn in the pen mode that just draws over everything that is underneath (the so-called copy mode).

```
::  FinishFigure Figure -> [DrawFunction];
    FinishFigure (RectF color rect) -> [SetPenMode CopyMode, DrawRectangle rect];
    FinishFigure points -> [];
```

The update function of the picture window. It redraws all figures in the program state. It is important that the figures are redrawn in the same order the user drawed them. The update area is ignored, so the whole picture will be re-drawn with every update.

```
::  Update UpdateArea State -> (State, [DrawFunction]);
    Update area state:(pos, color, fig, figs) -> (state, DrawFigures figs [SetPenColour color]);

::  DrawFigures [Figure] [DrawFunction] -> [DrawFunction];
    DrawFigures [RectF color rect | figs] drawfs
    -> DrawFigures figs [SetPenColour color, DrawRectangle rect | drawfs];
    DrawFigures [PointsF color points | figs] drawfs
    -> DrawFigures figs [SetPenColour color | DrawPoints points drawfs];
    DrawFigures [] drawfs -> drawfs;

::  DrawPoints [Point] [DrawFunction] -> [DrawFunction];
    DrawPoints [p | ps] drawfs -> [DrawPoint p | DrawPoints ps drawfs];
    DrawPoints [] drawfs -> drawfs;
```

EqPoint tests for the equality of Points.

```
::  EqPoint Point Point -> BOOL;
    EqPoint (x,y) (p,q) -> AND (= x p) (= y q);
```

The constants of the program to enhance comprehension and maintenance.

```
MACRO
    FileId  -> 1;           ColorId    -> 2;        ToolsDialogId -> 1;
        ToolsId  -> 11;         BlackId  -> 21;         FiguresId     -> 11;
        QuitId   -> 12;         WhiteId  -> 22;             RectItemId    -> 111;
                                RedId    -> 23;             PointsItemId  -> 112;
                                GreenId  -> 24;
                                BlueId   -> 25;
```

In connection with this example some remarks can be made about programming interactive applications in Concurrent Clean.

The structure of large and small interactive programs is based on the two major components of an interaction: the program state and the devices that engage in the interaction. It is good practice to define the program state in a separate module and to define access functions on all components of the program state: one should treat the program state as if it were an abstract type (even if it is not). The advantage is that when the program state changes, only the access functions need to be changed. When pattern matching is used on the program state throughout the program it is clear to see that the program state cannot be changed without rewriting the complete code.

When designing the interface it is a good idea to start with the menu system. The menu system is the most static of all devices. Its definition generally covers all operations the programmer wants to be available to the user. It is quite easy to get a prototype running, consisting only of the menu system and some of the vital functions. Starting from this framework the application can be extended step by step.

In applications that use multiple windows in more or less the same way (text editors, drawing programs etc.), it is a good idea to start first with one window only. Once the application works correctly with one window, it is easy to generalise the implementation for an arbitrary number of windows.

6 Annotations

A reduction strategy repeatedly determines which part of the graph is going to be reduced by which rewrite rule. By default Concurrent Clean uses the so-called *functional strategy*. If there are several rule alternatives for a particular function, these alternatives are tried in textual order; patterns are tested from left to right; evaluation of arguments is forced when it is tried to match an actual argument against a corresponding non-variable part of the pattern. As a consequence arguments are only evaluated if they have to be matched against a pattern specified in a rule, otherwise they remain unevaluated. So, Concurrent Clean uses an evaluation scheme which is common for many lazy functional languages.

In Concurrent Clean, the functional strategy may be locally influenced by the use of annotations. When this strategy encounters an annotation it changes its default reduction order which will influence the way in which a result is achieved. Changing the order is in particular important if one wants to optimise the time and space behaviour of the reduction process.

Currently, two kinds of annotations are possible in Concurrent Clean:
- *strictness annotations* to locally change lazy evaluation in *eager evaluation*;
- *process annotations* to define *interleaved evaluation* on the same or *parallel evaluation* on another processor.

Annotations are placed at a node or at a reference to a node and may be inserted in definitions of new types and in type specification of rewrite rules (*global annotations*) as well as in the right-hand-sides of the rewrite rules themselves (*local annotations*).

Syntax for Concurrent Clean annotations:

```
Node            =    [Annotation] Symbol {[Annotation] Arg}
                     ;
Annotation      =    '{' StrictAnnot '}'
                |    '{' ProcessAnnot '}'
                |    StrictAnnot
                ;
```

6.1 Strictness annotations

The syntax for strictness annotations is simple:

```
StrictAnnot     =    '!'
                ;
```

When a strictness annotation is encountered, the evaluation of the indicated subgraph is forced. This forced evaluation will also follow the functional strategy yielding a root normal form. After the forced evaluation has delivered the root normal form, the reduction process continues with the ordinary reduction order following the functional strategy. So, annotations let the reduction strategy deviate from the default functional evaluation order making the evaluation order partially eager instead of lazy.

Strictness annotations are in general used to force the evaluation of arguments of a function in the case that the function is known to be *strict* in these arguments. A function is *strict* in a certain argument if the evaluation of that argument on forehand does not change the termination behaviour of the program. Generally the evaluation of such a strict argument is needed to compute the function result. A strict argument can therefore be evaluated in advance.

Strictness annotations are important because, in general, functions with strict arguments can be implemented more efficiently (Brus *et al.* (1987), see also (Plasmeijer & van Eekelen (1993)). The largest gain in efficiency (one order of magnitude) is obtained for strict arguments of the basic types `INT`, `BOOL`, `CHAR`, `REAL` and `FILE`. Furthermore strictness annotations are also convenient if Concurrent Clean is used as an intermediate language for eager languages (such as Hope (Burstall *et al.* (1980)) or FP (Backus (1978))) instead of lazy languages such as Miranda (Turner (1985)).

Global strictness annotations in type rules

The strictness annotations in a type specification are called *global* because they change the reduction order for *all* applications of a particular function. Annotations in a type specification of a function are allowed to be placed before the type specification of either an argument on the left-hand-side or an argument of a tuple type appearing in a *strict context*. A tuple type is only in a strict context if it has been supplied with a (valid) strictness annotation or if it appears as the root of a right-hand-side. Strictness annotations are only valid if they appear in a strict context. Intuitively, such a strictness annotation indicates that the corresponding argument is always reduced to root normal form before the corresponding rule is applied.

Strictness annotations may also be used in strict tuple types appearing in a definition of a type. The fact that on the left-hand-side of a type definition no patterns are allowed implies that this situation can only arise in a type synonym definition for (an instance of) a tuple type. The meaning of these annotated synonym types can be explained with the aid of a simple program transformation with which all occurrences of these synonym types are replaced by their right-hand-sides (of course, annotations included).

Example of a global strictness annotation in type rules:

```
::   IF !BOOL  x    x        ->   x;
     IF  TRUE  then else     ->   then;
     IF  FALSE then else     ->   else;
```

Notice that the strictness annotation will force the evaluation of the first argument before the `IF` function itself is called. In the pattern matching phase it is guarantied that the first argument is already in root normal form such that the pattern matching can directly start with comparing symbols.

Example of a global strictness annotation in a tuple type:

```
TYPE
::   Complex                 ->   (!REAL,!REAL);

RULE
::   +C !Complex !Complex    ->   Complex;
     +C (r1,i1) (r2,i2)      ->   (+R r1 r2,+R i1 i2);
```

which is equivalent to

```
RULE
::   +C !(!REAL,!REAL) !(!REAL,!REAL) -> (!REAL,!REAL);
     +C (r1,i1) (r2,i2)               -> (+R r1 r2,+R i1 i2);
```

Example of an illegal global strictness annotation:

```
::   Hd  [!x]            ->   x;
     Hd  [x | y]         ->   x;
     Hd  []              ->   ERROR "head of an empty list taken";
```

Allowing tuples to be strict makes it possible to define functions that deliver multiple results eagerly (i.e. all results already evaluated). Note that in a lazy functional language the arguments of a tuple cannot be evaluated since the tuple constructor that 'glues' the results together forces a lazy context for these arguments. With a strictness annotation it is possible to overrule this laziness. The presence of strict tuple in a program can be used to generate very efficient code for it (Nöcker & Smetsers (1990)).

In reasoning about programs with global strictness annotations it will always be true that the annotated argument will be in root normal form when the corresponding rule is applied.

Local strictness annotations in rewrite rules

Strictness annotations in rewrite rules are called *local*. They change only the order of evaluation for a *specific* function application. These annotations can only appear in the right-hand-side of rewrite rules.

Before the evaluation continues after applying a rewrite rule all strict annotated nodes of the right-hand-side of the applied rewrite rule are evaluated. Strictness annotations in rewrite rules can be placed anywhere on the right-hand-side, both on arguments as well as on nodes. If a local strictness annotation is put on a node, this is equivalent with putting the annotation on each reference to that node.

Example of strictness annotations on the right-hand-side:

```
F x y -> -- !y IF x
      -> ++ !y;
```

In this function it is clear that a common part of the two guarded expressions can safely be reduced.

Each node on the right-hand-side is considered to be either *strict* (nodes that have to be evaluated to root normal form) or *lazy* (nodes that have not yet to be evaluated to root normal form). The following rules specify whether or not a particular node is lazy or strict (remember that putting a local strictness annotation on a reference to a node is equivalent with putting the annotation on that node):

1) the root node of the right-hand-side is strict;
2) the global annotated arguments of a strict node are strict;
3) a local annotated node is strict;
4) all the other nodes are lazy.

All strict nodes on the right-hand-side are evaluated to root normal form before the evaluation continues with the reduction of the (always strict) root node.

Example illustrating states of nodes:

```
::   F ![INT]      INT   -> [INT];
     F []           n    -> [];
     F [a | b]      n    -> [n | F b (G a n)];

     G a b               -> + a b;

     Start               -> r: F a b,
                            a: F e c,
                            e: [1],
                            c: ++ 5,
                            b: G 1 ! d,
                            d: ++ 4;
```

In the `Start` rule, the following nodes are strict: `r`, `a`, `e` and `d`; all the other nodes are lazy.

At first sight, a shared node might introduce conflicts because on one hand it might be lazy but on the other hand it might be strict. Consider for instance the following example:

```
     Start               -> (n, ! ++ n),
                            n: Fac 1000;

::   ++ ! INT            -> INT;
```

According to the first application of `n`, the corresponding node (with symbol `Fac`) is lazy. However, due to the local ! annotation the application of ++ is strict which makes its argument `n` also strict. It is clear that in such case it is safe to reduce `n` instead of building a graph for it. So, the node must be strict.

Fortunately, the definition above defines a node to be lazy *unless* it is strict. So, a conflict will never occur.

Strictness annotations and gain in efficiency

The following example illustrates the gain in efficiency that is achieved when strictness annotations are added to a Concurrent Clean program. Consider the following Concurrent Clean program which calculates the Ackerman function.

```
MODULE Acker;

IMPORT deltaI;

RULE
::   LazyAcker INT INT -> INT;
     LazyAcker 0 j -> ++ j;
     LazyAcker i 0 -> LazyAcker (-- i) 1;
     LazyAcker i j -> LazyAcker (-- i) (LazyAcker i (-- j));

::   Start -> INT;
     Start -> LazyAcker 3 7;
```

When this computation is performed on a MacII with 5Mb heap it will take 13.7 seconds + 0.1 seconds for garbage collection.

```
::   StrAcker INT INT -> INT;
     StrAcker 0 j -> ++ j;
     StrAcker i 0 -> StrAcker !(-- i) 1;
     StrAcker i j -> StrAcker !(-- i) !(StrAcker i !(-- j));

::   Start -> INT;
     Start -> StrAcker 3 7;
```

The same computation, but now with local strictness annotations added (as shown in the definition of `StrAcker`) will only cost 5.9 seconds + 0.0 seconds garbage collection. In the implementation the gain in efficiency is obtained because less nodes have to be built. Local annotations might increase the execution speed in certain cases as shown by the previous example. However, in general it is hard to predict whether or not this improvement will be significant.

```
::   Acker !INT !INT -> INT;
     Acker 0 j -> ++ j;
     Acker i 0 -> Acker (-- i) 1;
     Acker i j -> Acker (-- i) (Acker i (-- j));

::   Start -> INT;
     Start -> Acker 3 7;
```

A significant gain in efficiency is obtained when global strictness annotations are placed in type specifications (see the definition of `Acker` above). The same computation will now only take 1.5 seconds + 0.0 seconds garbage collection. The gain is one order of magnitude compared with the previous definitions. In fact, the strictness analyser (see below) incorporated in the code generator can derive automatically that for all applications of the function the evaluation of the arguments is needed, in particular for the recursive call itself. Because both arguments and result are of type `INT`, no nodes are made at all (hence no garbage collection is needed). The calculation is performed on a stack and registers are used where possible. The speed is comparable with a recursive call in highly optimised C or with the speed obtainable when the function was programmed directly in assembly. Generally, it is not necessary to add strictness annotations by hand. If speed is of vital importance and the strictness analyser (see the next section) is not able to detect them the programmer should use global annotations where possible. Note that for efficiency reasons many of the predefined library functions contain global strictness information.

6.2 Strictness analysis

Though strictness annotations can be inserted by a programmer, many of them can be found automatically by a strictness analyser. In general, it is undecidable whether or not a function is strict in its arguments. However, it appears that in many cases a lot of useful information can be found.

Some strictness annotations are very difficult or even impossible to detect. A good analyser is capable of finding many global annotations for the left-hand-sides of type specifications of rewrite rules. However, finding information for tuple types is almost impossible. Consider the example from section 1.5:

```
TYPE
::   Complex                  -> (REAL, REAL);

RULE

::   +C Complex Complex       -> Complex;
     +C (r1,i1) (r2,i2)       -> (+R r1 r2, +R i1 i2);
```

The only information that for the function `+C` can be derived is that it is strict in its arguments (because of the pattern matching). However, the result of the function is a tuple of which the arguments apparently don't need to be reduced. If strictness annotations are added to the type definition of `Complex`, all other strictness annotations can be derived. Note that the type of `+C` has to be specified for finding all strictness properties: the type checker cannot derive that the result is a `Complex` value.

The strictness analyser in the Concurrent Clean System is based on *abstract reduction* (Nöcker (1990,1992))**.** In abstract reduction a domain of sets of values is defined. Reduction in this domain means reduction of sets. Because this domain of sets is not finite fixed points techniques are not applicable. Problems due to recursive functions are solved with a technique called *path analysis*. With this method also other kinds of strictness can be derived, for example, strictness properties for functions over lists. (however, such information is not used by the Concurrent Clean compiler). It appears that this analyser can find much information. The analysis itself is quite fast. Consider the functions:

```
Append []    y      ->   y;
Append [a|r] y      ->   [a | Append r y];

Foldr op r []       ->   r;
Foldr op r [a|x]    ->   op a (Foldr op r x);

Catenate l          ->   Foldr Append [] l;
```

With strictness analysis based on abstract interpretation for the function `Catenate` a fixed point in a rather complex domain has to be determined (in the list case). With abstract reduction the right information is found quite easily (see Nöcker (1990) for the analysis).

6.3 Process annotations

In Concurrent Clean processes can be created by adding process annotations (van Eekelen (1988), van Eekelen *et al.* (1990)) to rewrite rules. Currently only *local process annotation*s can be specified in the right-hand-side of rewrite rules*.

If a process annotation is encountered, the evaluation of the indicated subgraph is forced as with a strictness annotation, following the functional strategy until a root normal form is reached. The important difference with strictness annotations is that with process annotations new reduction processes are created that perform the evaluation. These new reduction processes can run interleaved or in parallel with the

* The Concurrent Clean System only supports parallelism when the simulator is used. Parallel code can already be generated for a Parsytec Supercluster. For Macintosh and Sun a generator for parallel code is under development. At the moment parallel annotations are ignored when concrete code is generated.

original reduction process. The original process continues with the evaluation in the ordinary reduction order independently.

Process annotations let the reduction strategy deviate from the default functional evaluation order making the evaluation order partially eager instead of lazy. As with strictness annotations, process annotations force the evaluation of the indicated subgraph. Again one has to be careful not to change the termination properties of the program with adding process annotations. The result of a program with process annotations is *not* always the same as the result without annotations: the parallel processes reduce on copies of indicated subgraphs such that some sharing might be broken up. However, the output of a Concurrent Clean program is not printed as graph but as a term. When sharing is removed by making copies of all shared subgraphs (*unravelling*), the results are the same, i.e. if the normal forms are unravelled to terms, these terms are identical (Eekelen (1988)).

The process annotations of Concurrent Clean are especially designed to make efficient parallel evaluation on loosely coupled parallel machine architectures possible. A *loosely coupled parallel architecture* is defined as a multi-processor system which consists of a number of self-contained computers, i.e. sparsely connected processors each with private memory. An important property of such systems is that for each processor it is more efficient to access objects located in its own local memory than to use the communication medium to access remote objects. In order to achieve an efficient implementation it is necessary to map the computation graph to the physical processing elements in such a way that the communication overhead due to the exchanging of information is relatively small. Therefore, the graph to be rewritten has to be divided into a number of subgraphs (*grains*) indicating the parts of the program graph that can be reduced in parallel. A real speed-up on parallel architectures can only be achieved if redexes that yield a sufficient large amount of computation, are evaluated in parallel while the intermediate links are sparsely used (*coarse grain* parallelism).

It would be nice if the Concurrent Clean System would detect these coarse grains automatically. As with strictness, the complexity of a computation is in general undecidable. And even worse, also no satisfactory automatic approximation exists. Therefore process annotations have to be explicitly added by the programmer. In general only the programmer can decide which part of the program represents a large amount of reductions relative to the expected communication overhead. With the annotations in Concurrent Clean the programmer also specifies *which* information is copied *when* from one processor to another. In this way a specific program can be tuned for a particular parallel machine architecture.

As one might expect from the name of the language, process annotations are an important feature of Concurrent Clean. In Concurrent Clean there are currently two kinds of process annotations are available. There is an annotation to create a *parallel* process (with the `{P}` annotation) and an annotation to create an *interleaved* process (with the `{I}` annotation).

Syntax for process annotations:

```
ProcessAnnot  =    'I'
              |    'P' ['AT' Graph]
              ;
```

With the process annotations available in Concurrent Clean complicated parallel algorithms can be expressed. Synchronous as well as asynchronous process communication can be specified. Arbitrary communications between processes can be defined, as well as various kinds of process topologies (hierarchical and non-hierarchical). The currently available annotations can be seen as basic building blocks. In future versions of Concurrent Clean higher level primitives will be predefined based on the currently available building blocks to make the difficult task of parallel programming a bit easier.

Creation of processes on processors

When, in Concurrent Clean, a process annotation is encountered in the contractum pattern during a rewrite step, a new reducer is created with as task the evaluation of the corresponding sub-graph.

With the `{I}` annotation a new *interleaved reducer* is created on the *same* processor that reduces to root normal form (following the functional strategy). Such an interleaved reducer dies when the root normal form is reached. However, during the evaluation of this result other reducers may have been created.

With the `{P}` annotation a new so-called *parallel reducer* is created. This reducer is *preferably* located on a *different* processor working on a copy of the corresponding subgraph, but, when this is somehow not possible, it may be created on the *same* processor. Reducers that are located on different processors run in parallel with each other. Reducers that run on the same processor run interleaved. The semantics of `{P AT location}` are a small extension of the semantics of `{P}`. The intention of this annotation is to give the programmer some control over the distribution of the processes over the actual processor topology.

In this section the concept of *lazy copying* is very important: when a copy of a sub-graph is made for the evaluation of this graph by a parallel reducer nodes that are already being reduced by some other reducer will not be copied until the corresponding redex has been reduced to root normal form.

Locking

An interleaved or parallel reducer will be locked (suspended) if it wants to reduce a redex that is already being reduced by some other reducer. A locked reducer can continue when the redex has been reduced to root normal form. A reducer can demand the evaluation of a sub-graph located on another processor. Such a demand always takes place via a *communication channel* (a lazy copy node). If the channel is referring to a sub-graph in root normal form, a lazy copy of this subgraph is made on the processor such that it can be further evaluated by the demanding reducer. If the sub-graph is not yet in root normal form, the demanding process is locked. But, in that case there always will exist a reducer on the other processor that reduces the sub-graph to its root normal form.

So, all communication channels are always being *served* by processes to ensure that demanding processes are not locked unnecessarily.

Annotations and evaluation order

When there are several local annotations specified in a contractum, the order in which they have to be effectuated is in principle depth-first with respect to the sub-graph structure. The underlying philosophy here is that in many cases it makes sense to deal with arguments before the function is called. So, it is nice if a programmer can specify this using an annotation. In fact, also the definitions of `{P}` and `{I}` rely on this property.

Local and global annotations

An extra complication in Concurrent Clean with respect to the semantics of annotations is the fact that Concurrent Clean also allows global strictness annotations. It has to be defined in what order local and global annotations are effectuated.

The general idea here is that using local annotations a programmer can explicitly specify the order of evaluation in the right-hand-side of a rewrite rule. So, the choice has been made not to take global annotations into account when determining the evaluation order of a function. Of course, it still holds that before a function is actually called first its globally annotated strict arguments have to be evaluated. So, before the chosen function will actually be evaluated, first its globally annotated arguments will be reduced, after which the function will be evaluated according to its local annotations.

Maximising actual parallelism

When there is not a clear hierarchy between annotations, it would be nice if the degree of parallelism would be maximised in choosing between the various possibilities.

So, for these cases the choice has been made to do first `{P}`'s, then `{I}`'s, and finally `{!}`'s. These priorities also hold for the cyclic case and for the case in which several annotations are put on the same node. At first sight it may not seem to make much sense to even allow the latter case because only one reducer will be able to reduce the indicated sub-graph. All others will be locked until the reduction is finished. However, a user may want to specify such a behaviour in which e.g. first a parallel reducer is started on a node (with a `{P}`) and immediately the father process will wait for the result (due to a `{!}`).

However, parallelism can be even further maximised as is clarified with the following situation: suppose, that a local process annotation and global strictness annotation are both valid for the same node. Then, as defined above the priority lies with creating the child process first in order to maximise the parallelism. It can however be the case that the father process has again the choice between waiting for the child process to produce its result and for instance evaluating an other argument of the corresponding function call. Then, parallelism is only really maximised if the other argument is chosen first. Otherwise, the father process would just be waiting in parallel.

So, when arguments are equal in priority (e.g. they are either both globally strict annotated or both locally strict annotated) the choice is made to effectuate first those strictness annotations for which it is known that (due to an analysis of the right-hand-side of the corresponding rule) not yet another process is already evaluating the corresponding sub-graph. In this way unintended waiting is prohibited.

Example of maximising parallelism:

```
::   Fib INT   -> INT;
     Fib n     -> + ({P} Fib (-- n)) (Fib (- n 2));
```

The function + is globally strict in both its arguments. A child process is created that will reduce a lazy copy of the first argument in parallel. The father will continue with the second argument.

```
     Fib n     -> + (Fib (-- n)) ({P} Fib (- n 2));
```

But, when a child would be created on the second argument instead, the father process will evaluate the first argument. In this way a nice symmetric behaviour is obtained with a maximum amount of actual parallelism.

Specifying process locations

Processors in a distributed architecture are generally sparsely connected. When two parallel processes heavily communicate with each other it is efficient to locate them on processors that are directly connected via a hardware link. If one is aiming for an optimal performance the actual location of a process should be definable. For this reason there is a possibility in Concurrent Clean to specify the processor on which a parallel process preferably has to be created. The use of this possibility might make the efficiency of the program fully dependent of the actual machine the program is running on. So, the user is advised to refrain from the use of this feature as much as possible.

The `{P}` annotation can be extended with a *location directive* `{P AT location}`, where `location` is an expression of predefined type `PROCID` indicating the processor on which the parallel process has to be created. In the library deltaP (see appendix B) functions are given that yield an object of this type.

Some of the primitives can be used without detailed knowledge of the actual processor topology. For instance, with the function `NeighbourP` one can create a process n links (hops) away from another processor (e.g. the current processor `CurrentP`).

With the primitive `ITOP` one can exactly specify which processor is intended. The primitive converts an integer value to a value of type `PROCID`. It is assumed that each processor of a concrete architecture can uniquely be identified by an integer value. How this mapping is defined is implementation dependent. So, the programmer must know this mapping and the actual processor topology to make optimal advantage of it.

In any case, it holds that whatever the use of process locations is, the *result of the program cannot be influenced*. The only factor that can be changed by the use of process locations is the actual process allocation on the machine.

Specifying process structures

The fact that Concurrent Clean is a graph rewriting language is clearly an advantage for the specification of complex process structures. The graph structure can be used to explicitly define the demanded process structure. To illustrate the use of graph rewriting in specifying process topologies below several examples are given of Concurrent Clean specifications of various process structures and different kinds of process behaviour.

A hierarchical process topology

Divide-and-conquer parallelism is expressed with hierarchical process topologies. They have a straightforward process behaviour. This can be expressed very easily in Concurrent Clean.

A hierarchical process topology in Concurrent Clean.

```
::   Fib INT    -> INT;
     Fib 0      -> 1;
     Fib 1      -> 1;
     Fib n      -> + {P} left {P} right,    IF > n Threshold
                -> + left right,            IF > n 2
                   left:  Fib ! (-- n),
                   right: Fib ! (- n 2);

MACRO
     Threshold -> 10;
```

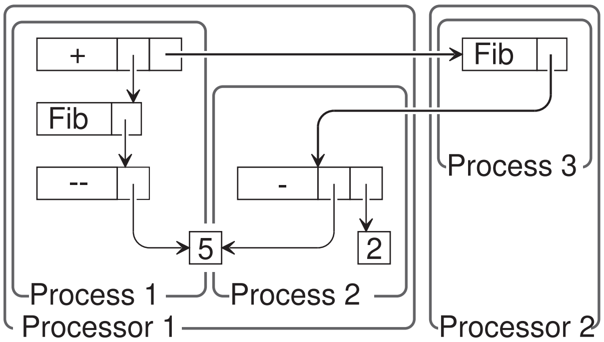
The `{P}` annotations specify that both calls of `Fib` can be evaluated in parallel on another processor. First `(-- n)` and `(- n 2)` are evaluated forced by the local strictness annotations because these annotations are put on sub-graphs of the graphs on which the process annotations are put. Hereafter two parallel processes are created on lazy copies of `right` and `left` respectively. The father process continues with the addition but it has to wait until the arguments are evaluated and have been lazy copied back.

A simple non-hierarchical process topology

In the following example a simple non-hierarchical parallel reducer topology is demonstrated. It serves the purpose of explaining how such parallel reducer topologies can be expressed.

The second argument of `Fib` will be executed by another parallel reducer but the argument of that call is reduced interleaved.:

```
::   Fib INT   ->   INT;
     Fib 0     ->   1;
     Fib 1     ->   1;
     Fib n     ->   + (Fib (-- n)) ({P} Fib ({I} - n 2 ));
```



A non-hierarchical parallel reducer structure: A snapshot of the program execution of `Fib 5` is given.

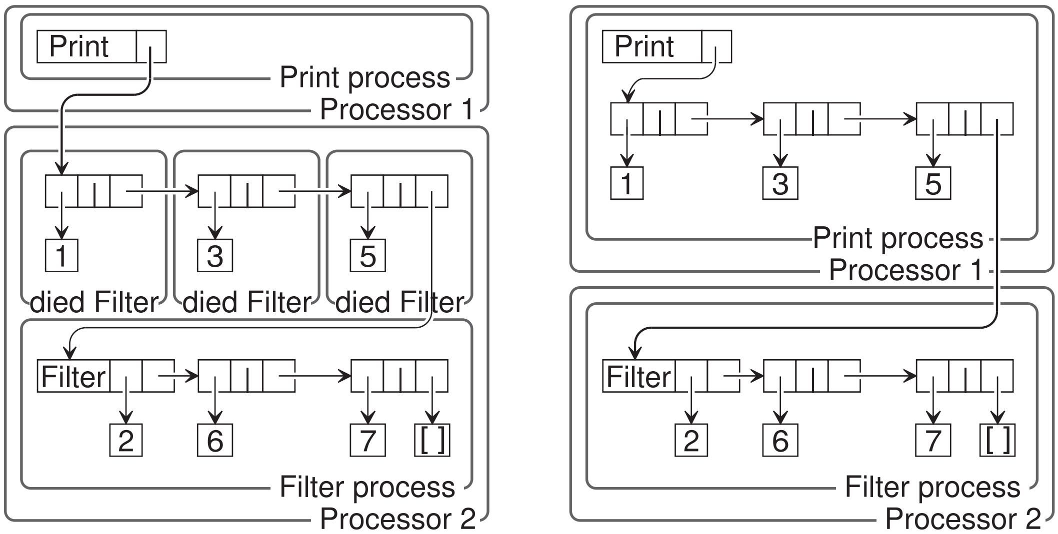
Asynchronous process communication with streams

To express an asynchronous communication stream a sequence of interleaved reducers has to be created. Each interleaved reducer computes a partial result that can be sent across a channel. Just before the reducer delivers its root normal form (and dies) it creates a new interleaved reducer that delivers the next result in the same way. When partial results are delivered more frequently than is demanded, all partial results (a *burst*) are sent at the same time. The lazy copy will copy all partial results at once but it stops at those nodes that are under reduction by a reducer. Note that cyclic graphs can be sent to another processor and that a parallel reducer can have several streams.

Example of an asynchronous communication behaviour with streams:

```
::  PrintList [INT]    ->  [CHAR];
    PrintList list     ->  Print ({P} Filter list 2);

::  Filter   [INT]     INT ->  [INT];
    Filter   []        p   ->  [];
    Filter   [f | r]   p   ->  Filter r p,    IF = (% f p) 0
                           ->  [ f | {I} Filter r p ];
```

The process reducing `PrintList` creates a new parallel reducer: `Filter`. This process removes from its first argument, which is a list, all the elements that are divisible by the number `2`. When `Filter` has computed an element of the result list it creates a new `Filter` process to compute the next number after which it dies. Several results may be added to the stream before they are communicated on demand. The end of the stream will be a channel referring to the current filtering process.



Snapshot of a filtering process working on a list containing the natural numbers from 1 to 7. The root normal forms yielded by successive interleaved `Filter` processes, can be shipped with one lazy copy action. The active interleaved `Filter` process acts as a copy stopper and a new channel is created to this process. If more results are demanded, the lazy copy action is continued as soon as new results become available.

The sieve of Eratosthenes is a classical example in which parallel sieving processes are created dynamically in a pipeline. In Concurrent Clean the sieve problem using !, `{P}` and `{I}` annotations can be expressed as follows:

```
::  Start     ->  [INT];
    Start     ->  Primes;

::  Primes    ->  [INT];
    Primes    ->  Sieve ({P} Gen 2);

::  Gen INT       ->  [INT];
    Gen Limit     ->  [ ];
    Gen n         ->  [ n | {I} Gen ! (++ n) ];
```

```
::   Sieve [INT]         ->   [INT];
     Sieve [ ]           ->   [ ];
     Sieve [ p | s ]     ->   [ p | {P} Sieve ({I} Filter s p) ];
```

This example is not much different from the example of section 1.9. The main difference is the local strictness annotation in the `Gen` rule. Because of this local annotation integers are passed to the `Sieve` reducers by the `Gen` process instead of a graph that will reduce to the wanted integer. The local annotation used here is a bit more efficient than the global annotation that was used in section 1.9, because there still one ++ node was sent from the `Gen` process to the `Sieves`.

Communicating work or data

When a reducer yields a root normal form it dies and the yielded sub-graph in root normal form can be lazy copied to a demanding process. The copying process will postpone copying on those parts of the graph that are under reduction by another process. This makes it possible to decide which part of the graph is lazy copied back. The sub-graph in root normal form generally will contain several redexes (work). When such a redex is returned in the lazy copy work is shipped from one processor to another. One can choose whether or not work should be locally calculated before it is shipped. In that case an interleaved process should be created on the corresponding redex before the root normal form is delivered. So, for each redex one has the choice how it should be handled.

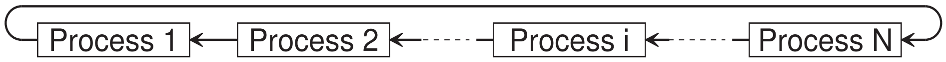
Choosing between data or work:

```
::   F (=> INT INT) BOOL     -> [ INT ];
     F g            h        -> Map g ({P} BigList h);

::   BigList BOOL            -> [ INT ];
     BigList h               -> [ 0 | Gen 2 ],  IF h
                             -> [ 1 | {I} CompList ];
```

The function `F` maps a function `g` over a list that is computed by a parallel process `BigList`. Depending on the evaluation of `h`, either a simple big list is returned or a big list that takes some computing power. In the first case work is copied back: not the list itself is returned but an expression is returned that computes this list when it is evaluated. In the second case data is copied computed by an interleaved process `CompList` that is created to compute the list before it is shipped.

Cyclic process topologies with location directives

The following example shows how a cyclic process topology, i.e. a number of parallel reducers that are mutually dependent, can be created.



A cyclic process structure.

The process structure as indicated in the figure above can be specified directly in the following way:

```
::   Start -> [INT];
     Start -> last: CreateProcs NrOfProcs last;

::   NrOfProcs -> INT;
     NrOfProcs -> 5;

::   CreateProcs INT    [INT]     -> [INT];
     CreateProcs 1      left
     ->   Process 1 left;
     CreateProcs pid    left
     ->   CreateProcs (-- pid) ({P} Process pid left);

::   Process INT   [INT]     -> [INT];
     Process n     left
     ->   [ n | {I} Process (+ n (Hd left)) (Tl left) ];
```

`CreateProcs` is initially called with a reference to itself thus forming a cyclic process structure of length one. `CreateProcs` recursively creates new processes `Process`. Each `Process` has a process number. `CreateProcs` will finally form the first `Process` in the cycle with process number `1`. Each `Process` is connected to the next one, i.e. the one with the next `pid` number, by means of a channel. During the creation of the processes this channel is passed as a parameter called `left`. `Hd` and `Tl` take in a lazy way respectively the head and the tail of the channel. Note that the type of `left` has to be the same as the result type of `CreateProcs` and `Process`. The function `Process` will in general produce a lazy stream of values (in the example integer values) that can be used as input by another `Process`. Due to the use of `{I}` annotations recursively a part of the stream can be produced as a result. In the example an infinite list of rapidly increasing numbers is generated (`[1, 3, 8, 20, 48, 112, …]`).

In the same way cyclic process structures can be explicitly specified in more elaborate examples.

In the cyclic example given above every process is heavily communicating with both its neighbours. So, it is probably more efficient to create a new process on a processor to which a direct link exists.

This can be done by replacing the `{P}` annotations above by the location directives `{P AT NeighbourP 1 CurrentP}`. However, this only guarantees that the new process will be located on a direct neighbour: it can happen that effectively only two processors are used. This can be avoided by making use of knowledge about the actual machine topology, e.g. using the location function `ITOP`.

Assuming that a neighbouring processor is found by taking a processor with identification one lower than the previous processor identification, the problem above can be solved as follows.

```
CreateProcs pid left
->   CreateProcs (-- pid) ({P AT ITOP pid} Process pid left);
```

Deadlock

One has to be very careful not to introduce deadlock when complicated process topologies are created, especially when several communication channels are being served by one and the same process. Complicated process topologies are often created by using mutually recursive functions or cyclic graph structures on which processes are created. There is a *deadlock situation* when processes mutually need locked information from each other in order to continue. None of those processes is then able to make any progress. This situation is similar to the sequential case of a cycle-in-spine error. More precisely: when all process annotations occur on positions that are needed for the sequential evaluation, the following holds:

If due to the process annotations a deadlock situation occurs, ignoring the process annotations would have led to a sequential cycle-in-spine error.

A deadlock example is given by a variant of the definition of the function `Process` in the cyclic process topology example of the previous section.

```
::   Process INT    [INT]     -> [INT];
     Process n      [hd|tl]   -> [ n | {I} Process (+ n hd) tl ];
```

A deadlock occurs since each process, due to the pattern match, wants the result of the previous process before it creates a result itself.

7 Changes and Additions

The description of Concurrent Clean in this manual corresponds with the Concurrent Clean System version 0.8.4. Besides the fact that some minor bugs have been repaired, the differences with version 0.8.3 are:

- The way to retrieve the current settings of a dialog has been improved. These settings are now represented by the new predefined abstract type `DialogInfo`, from which the settings of editable text fields, check boxes etc. can be retrieved by means of new predefined functions in the `deltaDialog` library module. As a result of this the `deltaDialog` and `deltaControls` library modules have changed. An overview of the functions defined in these modules can be found in appendix C. In the library module `deltaIOSystem` the new type `DialogInfo` has been added and the types `ButtonFunction` and `DialogFunction` have become:

```
::   ButtonFunction UNQ s UNQ io
     -> => DialogInfo (=> s (=> io (s,io)));
::   DialogFunction UNQ s UNQ io
     -> => DialogInfo (=> (DialogState s io) (DialogState s io));
```

- For compatablity with multi-user operating system the functions `HomePath` and `ApplicationPath` have been added to the `deltaSystem` library module. They prefix the filename given to them with respectively the full pathname of the home directory of the user and the full pathname of the directory in which the application resides.

The main difference between version 0.8.3 and version 0.8.1 is the I/O interface, which has been revised completely. The I/O interface is available now for Apple Macintosh systems and Sun4 systems under X Windows / Open Look™. The main differences with the I/O interface version 0.8.1 are listed below:

- The model on which the implementation of functional I/O in Concurrent Clean is based has been adapted to become a purely functional framework in which I/O can take place without ever violating the functional semantics. I/O is now done through the predefined `WORLD` environment, from which the sub-environments `FILES` (for file I/O) and `EVENTS` (for event I/O) can be retrieved. The `WORLD` environment is available as an optional parameter in the `Start` rule of a program. The `EVENTS` environment is used to create the `IOState` environment that was also used in previous versions of the I/O interface.

- It is now possible to define nested interactions: interactions that temporarily replace the current interaction.

- The possibility to close files has been added.

- The rather rudimentary dialog definition of version 0.8.1 has been replaced by a full-fledged implementation. This implementation makes it possible to define modal and modeless dialogs, as well as notices. All commonly used dialog items have been predefined: buttons, check boxes, radio buttons, pop up menus, editable text fields and static text fields. It is also possible to define one's own dialog items. A simple layout language is provided that is powerful enough to specify quite complex dialogs.

- The null device has been dropped. Instead it is now possible to define several timers (instead of just one), which may behave like the old null device by giving them a timer interval of zero.

- The implementation of windows has been extended. It is now possible to define fixed-size windows. Furthermore windows can be parameterized with a local cursor shape. The shape of the cursor (mouse pointer) can also be set globally.

- The mouse function now also accepts `ButtonDoubleDown` and `ButtonTripleDown` events.

- The modifier keys that are being held down are now passed to the mouse and keyboard function as a four-tuple of booleans instead of as a list of constructors.
- A full implementation for font handling has been added.
- The `deltaPicture` module (containing all draw functions) has been revised completely to become more straightforward, orthogonal and portable to other graphical interface systems.

The I/O interface is explained in full detail in chapter 5 of this manual. Appendix C contains a complete overview of the functions and types that have been predefined in the I/O libraries of Concurrent Clean.