Part 4
Implementation on sequential architectures

An efficient implementation of a functional language can be obtained in many ways. We shall not give a general overview of the several available methods (e.g. Landin, 1964; Turner, 1979a; Augustsson, 1984; Peyton Jones, 1987). Instead, we shall describe *one* implementation method in detail (Smetsers *et al.*, 1991). We think it is highly illustrative to study one particular implementation starting from the level of a particular full-featured programming language, leading through intermediate levels (see Figure P4.1) and finally ending with concrete machine code. Furthermore, the described implementation method is state-of-the-art and has many aspects in common with other methods.

So an implementation is described for a particular functional language (Miranda) based on the model that we consider to be the most suited for this purpose: FGRSs. First, the language is translated into an intermediate language (Clean) based on FGRSs. Since the level of Clean is still rather high compared with the level of a concrete machine, a Clean program is translated into code for an abstract stack-based machine (the ABC machine). The abstract machine forms an additional more concrete intermediate level enabling a relatively easy production of interpreters and code generators for various target machines.

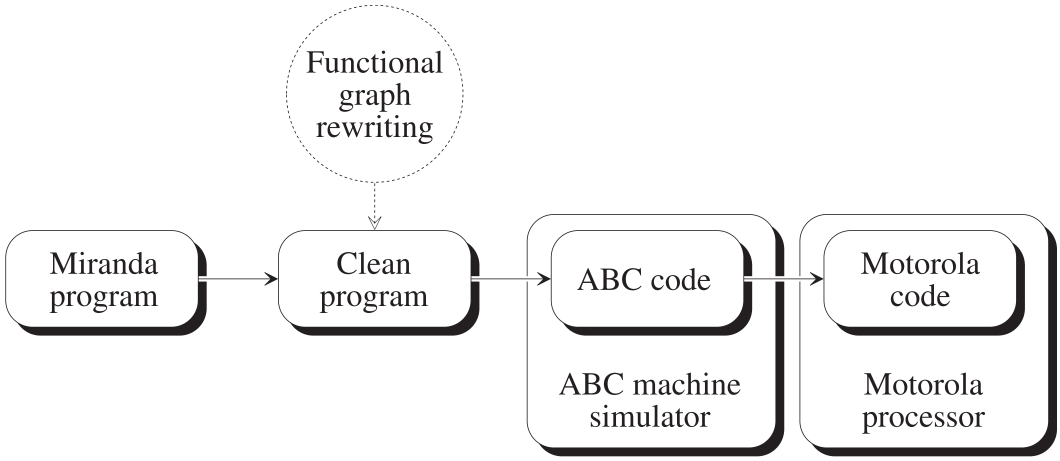

Figure P4.1 Two intermediate levels; three translations.

Owing to the intermediate levels, some of the information present in the original program might get lost, so some loss of efficiency might be introduced. This conceptual loss is generally well compensated since the compiler becomes more structured and easier to maintain.

Five chapters treat the two intermediate levels and the three corresponding translations. These transformations are also used in the Concurrent Clean system: the software package that goes with this book.

Chapter 8
Clean

Clean is an experimental lazy higher order functional programming language based on functional graph rewriting systems (FGRSs) (see Chapter 5). So Clean is a functional language in which computations are expressed in terms of graph rewriting. Clean is originally designed as an intermediate language to be used in the compilation path from (eager as well as lazy) functional languages to concrete machine architectures (Figure 8.1).

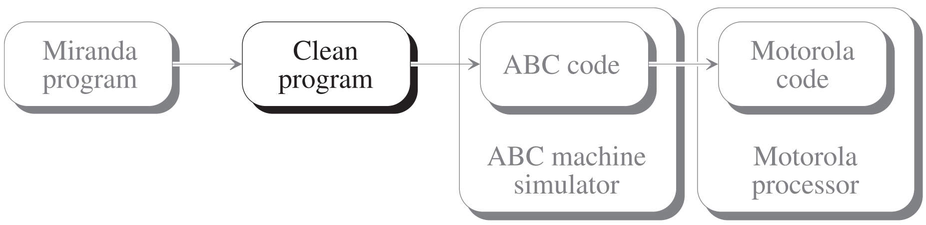

Figure 8.1 Clean as intermediate language.

In contrast with most other functional programming languages Clean supports only the most basic aspects of functional programming (see Chapter 1). Syntactical sugar (such as infix notation or *ZF*-expressions) is mainly excluded. Clean basically just realizes FGRSs syntactically extended with *macros* and *guards*. However, many additional semantic features have been added enabling an efficient and usable implementation, e.g. a *type system*, *annotations* to control evaluation order, a *modular structure* and *type attributes* enabling destructive updates of objects, as well as a library that enables a high-level specification of *input* and *output*.

Type information is important for programmers since it will help them to design programs which are correct and understandable. But, type information is also important because it can be used to generate efficient code. Furthermore, the type information can be used to create a modular environment. The type system of Clean is based on the type assignment system for TRSs as explained in Section 6.3. It includes polymorphic types, abstract types, algebraic types and synonym types, as well as basic types (see Section 8.2). Lists, tuples and function type constructors are predefined.

With *annotations* the evaluation order of a Clean program can be influenced by the programmer: the evaluation can be made (partially) *eager* instead of lazy. In this way a better space and time behaviour of functional programs can be obtained in a machine-independent manner (see Section 8.3). For efficient solutions of some problems the ability to share computations and to control the evaluation order is essential. In the extended language, Concurrent Clean (see Chapter 15), even parallel evaluation can be specified by the programmer.

Clean's *modular* structure (see Section 8.4) consists of separate implementation modules and definition modules, including a facility to import definitions from other modules implicitly and explicitly. There are predefined modules (libraries) for basic operations (δ-rules) on objects of basic types (integers, reals, characters, strings and booleans).

The interaction with the outside world has always been hard to incorporate in the functional framework. The Clean type system is extended with the *unique type attribute* that can be assigned to an object of any type (see Section 8.5). An object with the unique type attribute that is passed as an argument to a function can, under certain conditions, be reused. Referential transparency is still guaranteed while destructive updates of arrays, files and the like are made possible, resulting in an important efficiency gain.

Owing to these unique types one can specify with the Clean I/O library I/O devices (dialogs, menus, windows and the like) on a very high level of abstraction (see Section 8.6). As much as possible of the low-level I/O handling is done automatically by the library system. Predefined algebraic data types can be used to specify which devices should be activated and how these devices should be initialized. A programmer just specifies which (higher order) function is called when a certain part of an active device is triggered (e.g. a particular menu item has been chosen).

Using Clean as intermediate language has made 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. Since Clean is an experimental language, it has been equipped with some special features that are not found (yet) in other functional languages. Major and minor changes and extensions can be expected in future versions of the language.

8.1 Clean, a language for functional graph rewriting

Clean (Brus *et al.*, 1987; Nöcker *et al.*, 1991b; Van Eekelen *et al.*, 1992) was designed at the University of Nijmegen and it stands for: **C**lean **Lean**[1]. Clean is based on functional graph rewriting systems (see Chapter 5).

8.1.1 Basic syntax of rewrite rules

Clean uses the shorthand notation of graph rewrite rules. Apart from some additions for typing and for programming convenience (see Appendix B for a full syntax description of the language) the only syntactic differences from FGRSs are:

- The rule alternatives of a function definition are grouped together.
- A ';' marks the end of each rule alternative.
- Redirections consist of a single node-id only.
- The keyword `RULE` indicates that a sequence of graph rewrite rule definitions follows (so that they can be distinguished from other kinds of declaration).
- '->' is used instead of an $\rightarrow$.
- Empty nodes cannot be denoted in Clean.
- Several functions on basic types are predefined (see Section 8.1.3).
- Guards are added for choosing between rule alternatives (see Section 8.1.4).
- Lists and tuples are predefined data structures (see Section 8.1.5).
- Currying is allowed. A curried application is transformed at compile-time to an application of an internal 'apply rule' (see Section 8.1.6).
- Macros are used for programming convenience (see Section 8.1.7).

8.1.2 Semantics

Clean's basic semantics are treated in Chapter 5 (semantics of FGRSs). There are, however, some differences between FGRSs and Clean:

- When at run-time an empty node is created, in Clean an error message is given (see Section 5.8).
- Clean rules are typed (see Sections 6.3 and 8.2).

---

[1] Lean (Barendregt *et al.*, 1988) stands for the **L**anguage of **E**ast **A**nglia and **N**ijmegen, jointly designed at both universities. Lean has been used to experiment with *generalized* graph rewriting systems (see Section 5.9).

- In Clean, the functional strategy can be influenced by using annotations that make the evaluation partially eager instead of lazy (see Section 8.3).
- Input/output is treated in a special way. In order to communicate with the outside world, the Start rule optionally can have a parameter of type UNQ WORLD, representing the environment in which the program is called (see Sections 8.5 and 8.6).

8.1.3 Predefined functions and basic types

For practical reasons it is convenient that rules for performing arithmetic on objects of basic types (integers, reals etc.) are predefined. Then they can be implemented efficiently with the concrete representation and corresponding instructions available on the concrete machine.

```
<<   The factorial function in Clean using predefined functions.
>>
RULE
:: Fac INT      ->   INT;                  ==   type definition
   Fac 0        ->   1;                    ==   the factorial function
   Fac n        ->   * n (Fac (– – n));

:: Start        ->   INT;                  ==   type definition
   Start        ->   Fac 20;               ==   the initial expression
```

As in FGRSs prefix notation is used. Rules starting with '::' are type rules (see Section 8.2). Comments are placed between '<<' and '>>' or after a '=='.

Below, a list is given of several predefined functions (for a full list of the functions in the Clean library, see Appendix B).

Basic type	Examples of denotations	Examples of predefined functions
INT	1, 2, 2147483647	+, –, =, >=, AND%
REAL	1.5, 0.314E10	+R, –R, =R, >=R, SIN, EXP
CHAR	'a', 'b', 'c', '\n', '\007'	=C, >=C
BOOL	TRUE, FALSE	NOT, AND, XOR, =B
STRING	"This is a string \n"	+S, –S, =S, >=S, SLICE
FILE		StdIO, FOpen, FWriteC

8.1.4 Guarded expressions

In Clean, guards can be added to rewrite rules. A guard is a Boolean expression that can be seen as an extension of the pattern matching mechanism: a rule alternative is only chosen when its pattern matches a subgraph *and* its guard evaluates to TRUE. Pattern matching always takes place before the guard is evaluated. A guard is separated from the right-hand side of a rule by an optional comma and the keyword IF. In a guard the *infix* operators || (conditional OR) and && (conditional AND) can be used. The arguments of the operators are evaluated from left to right, and parentheses can be used to specify the priorities. Arguments of operators are not evaluated when their value is of no influence on the value of the result. For instance, when the first operand of || evaluates to TRUE, the second operand will not be evaluated: the result will be TRUE anyway.

Rule alternatives that have the same left-hand side (but different guards) can be grouped together 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).

Example of the use of guards in Clean:

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

8.1.5 Predefined data structures and type constructors

To provide some programming convenience, denotations for *lists* and *tuples* are predefined.

Lists

Lists are denoted as follows: Nil is denoted as [ ], Cons h t as [h | t] and the list with elements 1, 2, 3 and 4 is denoted as [1, 2, 3, 4]. Furthermore, [e_1, ... , e_n | r] denotes a list r prefixed with elements e_1 ... e_n. If T is a type, than [T] denotes the type *list of* T.

Merge merges two sorted lists of integers, removing double occurrences:

```
:: 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 ];
```

Tuples

Tuples and tuple type constructors are denoted as in Miranda. If T_1, ... , T_n are types, then (T_1, ..., T_n) denotes the type *tuple of* T_1, ..., T_n.

Example of Clean tuples and their types:

```
(3, 0, 'a')        ==  3-tuple of type (INT, INT, CHAR)
([0, 1, 2], +)     ==  2-tuple of type ([INT], => INT (=> INT INT))
```

In order to simplify the use of tuples a special syntactical construct is added that automatically generates projection functions. One can specify a definition of a number of node-ids simultaneously by placing a tuple of node-ids before a colon symbol in a node definition. The basic idea of this construct is illustrated by the example below:

Instead of

```
G …  ->   H a b,
          a: First g,
          b: Second g,
          g: F …;

F …  ->   (x, y);

First   (a, b)  ->  a;
Second  (a, b)  ->  b;
```

one can define G more compactly (First and Second are no longer needed):

```
G …  ->   H a b,
          (a, b):   F …;
```

The meaning of such a construct is that all these node-ids are bound at run-time to parts of the graph on the right-hand side of the colon symbol. This only makes sense if this graph denotes a function that delivers a tuple. The type checker (Section 8.2) verifies that this is true.

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 ];
```

Besides the benefit of notational elegance, the list and tuple constructs have the advantage that they indicate special predefined cases that can be implemented more efficiently than the fully user-defined data structures. In the case of the tuples it might even lead to fewer reductions since it is not necessary any more to apply a selector function explicitly to access the elements of the tuple.

8.1.6 Currying

Clean conceptually uses the functional notation (see Section 4.1.4). As explained in Sections 4.8 and 6.3.2, curried applications of functions can be realized by introducing additional rewrite rules that transform curried applications into uncurried ones. This method has a disadvantage. Many of these additional 'AP-rule' alternatives are needed: one alternative for each function that is used in a curried manner. It is much more convenient if this curry conversion is performed automatically.

This automatic conversion has been realized in Clean in the following way: a generic internal rule _AP is predefined (internal here means that this _AP rule cannot be used by the programmer explicitly) with which curried applications of any symbol F are transformed into uncurried applications at compile-time.

The Twice function as defined in Clean is given below.

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

The curried applications of f are internally transformed leading to

```
   Twice f          x  ->   _AP f (_AP f x);
```

The internal rewriting code for _AP provides that when a symbol is supplied with a sufficient number of arguments it will be rewritten according to the rule that corresponds to the curried symbol. This allows the programmer to use symbols with an arbitrary arity or to apply node-ids to arguments. If F is defined with arity *n*, say F a_1 a_2 ... a_n -> a_r, then _AP (... (_AP (_AP F a_1) a_2) …) a_n reduces to F a_1 a_2 … a_n. If there are too few arguments the expression is in root normal form (_AP is a partial function). Since the conversion of curried applications into uncurried ones will be done automatically there is no need at all for the programmer to be aware of the presence of this internal _AP rule.

This facility implies the need for a special predefined type constructor, the **curried function type**, denoted by =>. This predefined type can be compared with the function type constructor 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). Consequently, the type axioms of the Clean type system include the equivalence of

:: F -> => t_1 (=> t_2 (=> ... (=> t_n t_r) ...)) ; to :: F t_1 t_2 ... t_n -> t_r ;

if F is defined with arity *n*. The internal function _AP is typed as:

```
:: _AP (=> a b) a -> b ;
```

The first argument of _AP (i.e. the function that has to be applied) always has to be evaluated in order to compute the result. So _AP is strict in its first argument.

With this syntactic sugar the Hamming example of Chapter 5 can be specified in Clean in the following way (with the IMPORT statement, general δ-rules for arithmetic are imported (see Section 8.4)):

```
MODULE Hamming;
IMPORT delta;
RULE
:: Ham ->     [INT];
   Ham ->     x: [ 1 | Merge (Map (* 2) x) (Map (* 3) x) ];

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

8.1.7 Macros

In Clean one can define **macros**: macros are special rewrite rules that are rewritten (*expanded*) at compile-time. Macros can be used to assign a meaningful name to a constant or to an expression (graph). They can reduce the number of function calls at run-time. Furthermore, macros can be used as a named constant in a redex pattern.

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. Macros 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 macros in Clean.

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

8.2 The type system

Clean is a typed language conforming to the type system introduced in Section 6.3. So, in many cases, types in Clean are similar to but not exactly the same as those in Miranda. Note that in Clean each symbol is defined with a fixed arity. In Miranda functions are always curried and have at most one argument. As already explained in Section 8.1.4, the function type => (in prefix notation) is used for curried functions. Another difference is that identifiers are used for type variables instead of *s. Two major differences concern abstract types (in Clean abstract types are connected to the module structure; see Section 8.2.2) and typing patterns (see Section 8.2.3).

As in Miranda, types do not have to be specified explicitly by the programmer. For programmers in a hurry, types can be inferred by the Clean compiler if type information is left out. However, the programmer is strongly advised to give an explicit specification of the types.

As a special facility of the Clean system, type checking can be switched off. In that case specified types are not checked but *assumed* to be correct. When the type of a function is not specified at all, the most general type is assumed. This facility of Clean can be used to experiment with GRS rules that are not typeable in the usual system. It is also useful when Clean is used as intermediate language for an untyped functional language, such as SASL (Turner, 1979b). Of course, its use is very dangerous and it will generally lead to inefficient code.

8.2.1 Type definitions

The objects that are manipulated in Clean are graphs. With a type definition one specifies certain restrictions imposed on the graphs on which rewrite rules are defined. Syntactically, graph rewrite rules are used to define types. Semantically there is of course a great difference between type rules and rewrite rules. In fact, in Clean types specify restrictions on the underlying term structure, rather than on the specific graph structure. Therefore, all graphs in type definitions have to be *d*irected *a*cyclic *g*raphs (**dags**) without explicit node definitions.

Types are defined in a TYPE block, just like rewrite rules are defined in a RULE block. Blocks may be specified in any order. Each type rule must be preceded by a '::'. The function symbol of the type rule is called the **type symbol**. Each time the *same* **type variable** is used in a particular type rule, it stands for the *same* type.

8.2.2 Defining new types

As in Chapter 2 there are three kinds of type definition that introduce new types: *algebraic* type definitions, *synonym* type definitions and *abstract* type definitions.

Algebraic types

Algebraic types are specified by means of a type rule where each alternative has a right-hand side with a *fresh* root symbol: the **constructor**. All the arguments of the constructor are type instances. A **type instance** is either a type variable or a dag with a type symbol, as root of which all the arguments are type instances. The alternatives of an algebraic type definition can be separated either by an arrow symbol ('->') or by a vertical bar ('|').

Compare the following (polymorphic) algebraic type definitions with the corresponding definitions given in Chapter 2:

```
TYPE
:: Day      ->   Mon  | Tue | Wed | Thu | Fri | Sat | Sun;

:: Intlist  ->   NilList
            ->   IntList INT Intlist;

:: Tree x   ->   NilTree
            ->   NodeTree x (Tree x) (Tree x);
```

Synonym types

Synonym types permit the user to introduce a new name for an already existing type. Mutual dependencies of synonym type definitions are prohibited.

Example of a type synonym definition in Clean:

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

Abstract types

In Clean, *abstract type* definitions are only allowed in definition modules (see Section 8.4). Abstraction and hiding are considered to be two sides of a coin. So the module structure is used to hide the concrete signature of the abstract type to the outside world. An abstract type is actually a synonym type or an algebraic type of which the concrete definition is hidden in the corresponding implementation module. In the definition module only the left-hand side of the type definition is shown as well as the type of the functions defined on the abstract data structure (the abstract signature). To distinguish an abstract type definition from an ordinary type definition a special kind of type block is provided: an ABSTYPE block.

A well-known example of an abstract data type is given below:

```
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 corresponding implementation module:

```
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 predefined types can be seen as special cases of abstract types.

8.2.3 Type specifications for functions

The type of a rewrite rule is either explicitly specified or it is inferred by the system. When it is explicitly specified the type specification must precede the corresponding rewrite rule. The type rule consists of one rule; free type variables may appear on the right-hand side.

Using free type variables on the right-hand side of a type definition for a rewrite rule: F has as result type any list.

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

Typing partial functions

Some type errors cannot be detected at compile-time. A partial function can be 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 value.

At run-time the application of F in the start rule will not match. So F 1 will not yield the required type: INT. Hence, at run-time a type error is generated:

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

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

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 failure and safe rewrite rules are being used (see also Section 6.3.2). In FGRSs there is no special situation when a function application does not match any of the rewrite rules of the function: the result is a root head normal form. In Clean this is also the case as long as no type conflicts occur and functions are safely typed (see Section 6.3.2). So in Clean functions can appear on a 'constructor position'. In Miranda a non-matching function will always result in a run-time error: Miranda is a function constructor system.

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. But, the resulting graph is still correctly typed and has type Num. In Clean Succ and Pred have to be defined twice, once as function and once as constructor. Notice that the definitions of the functions Succ and Pred are safe according to the condition given in Section 6.3.2.

8.3 Strict annotations

By default Clean rules are reduced using the functional strategy. This strategy may be locally influenced by the use of *annotations* added to the Clean program. Annotations can be considered as parameters of the strategy function. They are very important if one wants to optimize the time and space behaviour of the reduction process.

In this section the **strict annotations** are discussed that can be added in Clean to the rewrite rules and to type rules with the effect that sometimes partially *eager* evaluation is used instead of *lazy* evaluation. Strict annotations are important because, in general, functions with annotated arguments can be implemented very efficiently (see Chapters 10 and 11). The largest gain in efficiency is obtained when arguments of the basic types are being annotated. Strict annotations are also convenient when Clean is used as an intermediate language for eager languages (such as HOPE (Burstall *et al.*, 1980) or FP (Backus, 1978)).

Strict annotations and the order of evaluation

Annotations let the reduction strategy deviate from the default functional evaluation order, making the evaluation order partially eager instead of lazy. Annotations can be placed in definitions of new types and in type specifications of rewrite rules (*global annotations*) as well as in the right-hand sides of the rewrite rules themselves (*local annotations*). If a strict annotation is specified, 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.

Strict annotations can only be added safely when it is known that the termination behaviour of the original program is not changed.

> A strict annotation might change the termination behaviour of a program: before the rewrite rule for F is applied, the reduction of the argument of F is forced due to the specified strict annotation. As a consequence, the normal form 1 of F W will not be reached any more.

```
:: Start    ->    INT;
   Start    ->    F !W;

:: W        ->    INT;
   W        ->    W;

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

Strict 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 (see Chapter 7). A strict argument can safely be evaluated in advance.

Ideally, a programmer should not be bothered at all with annotations to gain efficiency. As shown in Chapter 7, a good strictness analyser (such as incorporated in the Clean system) can detect many cases where strict annotations can be put safely. Therefore, user-defined strict annotations are generally only used when an ultimate gain in efficiency is demanded.

8.3.1 Global strict annotations in type rules

The strict 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 an argument on the left-hand side of the type rule.

Example of a global strict annotation in type rules:

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

Note that the strict annotation will force the evaluation of the first argument before the Cond function is examined as a candidate for rewriting.

In reasoning about programs with global strict annotations it will always be true that the annotated argument is in root normal form when the corresponding rule is applied. In the pattern matching phase this knowledge can be used: one can start by comparing symbols.

Another example of a global strict annotation: the Nfib function calculates the number of times it is called. Nfib is frequently used to determine the number of function calls per second for a particular implementation.

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

Strict annotations can also be placed in type rules to indicate that the corresponding parts of an argument or a result will always be evaluated when the object as a whole is evaluated.

Example of global strict annotations in tuple types:

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

Strict annotations may be used in the same manner in a type synonym definition. 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 (annotations included).

Example of (partially) strict tuple types in a type synonym definition. The definition of +C is equivalent to the definition given above.

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

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

Allowing tuples to be strict makes it possible to define functions that deliver multiple results eagerly (i.e. all results already evaluated). This is in contrast with the general lazy case where the evaluation of the arguments of a tuple is not forced since the tuple constructor that 'glues' the results together induces a lazy context for these arguments. With a strict annotation it is possible to overrule this laziness. The use of (partially) strict data types can lead to very efficient code (Nöcker and Smetsers, 1993).

8.3.2 Local strict annotations in rewrite rules

Strict annotations in rewrite rules are called **local**. They change only the order of evaluation for a *specific* function application. Local strict annotations can be placed anywhere on the right-hand side of a rewrite rule, both on arguments as well as on nodes. If a local strict annotation is put on a node, this is equivalent to putting the annotation on each reference to that node. When a rewrite rule is applied, all strict annotated nodes of the right-hand side of the applied rewrite rule are evaluated before the evaluation continues.

> Example of strict annotations on the right-hand side. For this particular occurrence of Cond (as defined in a previous example) it is indicated that a common part of the then and else part can be reduced safely.

```
F x y -> Cond x !y (++ !y);
```

Each *node* on the right-hand side is considered to be either **strict** (appearing in a **strict context**: it has to be evaluated to root normal form) or **lazy** (appearing in a **lazy context**: not yet to be evaluated to root normal form). The following rules specify whether or not a particular node is lazy or strict:

(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.

Before reducing the root node of the right-hand side all strict nodes other than the root node are evaluated to root normal form.

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

In the Start rule, the nodes r, a, e and d are strict; the others are lazy.

8.3.3 Strictness analysis

A strictness analyser is incorporated in the Clean system. This analyser is based on *abstract reduction* (see Chapter 7). The analyser can often determine strictness and is quite fast. When a function is found to be strict in a certain argument, a global strict annotation is generated automatically. Certain strictness information is very difficult to detect. Finding information for tuple types is almost impossible since it would require an analysis of all the possible uses of the type.

Consider again the following example:

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

The only information that can be derived for the function +C is that it is strict in its arguments (because of the pattern matching). The result of the function is a tuple that is in root normal form. One generally cannot conclude that the tuple arguments are needed and hence they cannot be reduced in advance.

8.3.4 Strict annotations and gain in efficiency

The following example illustrates the gain in efficiency that is achieved when strict annotations are added to a Clean program. Consider the following Clean program that calculates the Ackermann function.

```
:: 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;
```

Performing this computation on a MacIIx with a 4Mb heap takes 16.3 s + 0.2 s for garbage collection.

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

The same computation (StrAcker 3 7), but now with local strict annotations added (as shown in the definition of StrAcker) will only cost 9.6 s + 0.1 s for garbage collection. The gain in efficiency is obtained because fewer nodes have to be made. In general, it is hard to predict how much gain will be achieved.

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

A significant gain in efficiency can be obtained when strict annotations are placed in global positions (see the (type) definition of Acker above). The same computation (Acker 3 7) will now only take 1.9 s + 0.0 s garbage collection. The gain is one order of magnitude compared with the previous definitions. It is now known that for *all* applications of the function the evaluation of the argument is needed, in particular for the recursive call itself. As will be explained in Chapter 11, no nodes are made at all (hence no garbage collection is needed) because both argument and result are of type INT. The calculation is performed on a stack and registers are used where possible. The speed is comparable with a recursive call in highly optimized C or with the speed obtainable when the function is programmed directly in assembler. The example already reveals the cost of lazy evaluation and pure graph reduction compared with eager evaluation using stacks and registers. Fortunately, with the help of strictness analysis, much strictness information can be found automatically (for instance in the case of the Ackermann function defined above).

8.4 Modules

For a practical useful language separate compilation is a must. A modular structure enables this with the facility to hide actual implementations of types and functions. This can also be used to specify abstract data types (see also Section 8.2). The Clean modular structure very much resembles the Modula2 approach (Wirth, 1982) with one major exception: Clean's implicit imports, which simplify the definition of modules that just collect and pass-through definitions of other modules.

A Clean program consists of *definition modules* and *implementation modules*. In general, each definition module has a corresponding implementation module. An implementation module and its corresponding definition module bear the same module name. Each module is stored in a separate file with the same name as the module, with an appropriate extension (e.g. name.icl, name.dcl).

8.4.1 Implementation modules

An *implementation module* consists of a sequence of type and rule definitions. The types and rules defined in such an implementation module have in principle only a *local scope*: they have meaning only in the implementation module in which they are defined.

Example of an implementation module:

```
IMPLEMENTATION MODULE ListOperations;
FROM deltaM IMPORT ABORT;
RULE
:: Hd   [x]       ->   x;
   Hd   [ x | y ] ->   x;
   Hd   x         ->   ABORT "head of empty list ?! ";

:: Tl [x]         ->   [x];
   Tl [ x | y ]   ->   y;
   Tl x           ->   ABORT "tail of empty list ?! ";

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

An executable Clean program should have an implementation module with a Start rule declared in it. This module is called the **start module**.

8.4.2 Definition modules

In a *definition module* one can specify which types and rules (defined in the corresponding implementation module) are **exported**, i.e. are made visible to the outside world. To export a type or a function from an implementation module one has to *repeat* its type definition in the corresponding definition module. All other definitions in the implementation module remain hidden.

Example of a definition module:

```
DEFINITION MODULE ListOperations;
RULE
:: Hd    [x]      ->   x;
:: Tl    [x]      ->   [x];
:: IsNil [x]      ->   BOOL;
```

An abstract type can be realized by exporting the name of an algebraic or synonym type, but not its concrete realization (see also Section 8.2.2). A definition module does not need to have a corresponding implementation module. This can be useful for defining types on a global level. Such a definition module can also serve as a 'pass-through' module (see Section 8.4.3).

System modules

System modules are special definition modules indicated by the keyword SYSTEM instead of DEFINITION. This indicates that the corresponding implementation module does not contain ordinary Clean rules, but abstract ABC machine code instead (the ABC machine is explained in Chapter 10). This facility makes it possible to program directly in ABC code. This can be used to increase execution speed of heavily used functions or complex data structures. On demand, the Clean compiler will substitute the ABC 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. An advantage of system modules is that a certain increase of efficiency can be obtained in a device-independent manner, since ABC code is device independent. It goes without saying that for ABC programming advanced knowledge is needed of the code generation schemes and the run-time behaviour of the ABC machine. Imprudent programming of system modules will generally lead to unpredictable errors. Typically, predefined δ-rules are implemented as system modules (see Appendix B).

8.4.3 Implicit and explicit importing

Via an *import* statement an exported definition can be used in another implementation or definition module. Imported types and rules can be regarded as being predefined. There are two kinds of import statement: *explicit* and *implicit* imports.

Explicit imports

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.

Example of the use of explicit IMPORT statements: the sieve of Eratosthenes, which calculates prime numbers.

```
IMPLEMENTATION MODULE Eratosthenes;
FROM deltaI IMPORT ++, =, % ;
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) ];
```

Implicit imports

Implicit imports are import statements in which only the imported module name is mentioned. In this case *all* symbols that are **declared** (i.e. defined or imported) in the imported module are imported (and hence also declared in the module that contains the import statement). So not only are all symbols exported by the indicated module imported, but also all symbols which in their turn are imported in that definition module. In this aspect the Clean modular structure differs from the Modula2 approach. So with one simple IMPORT statement all related declarations are imported implicitly. Consequently, definition modules can serve as a kind of 'pass-through'.

All delta-rules can be imported easily by importing the module delta via the implicit import statement: IMPORT delta;. This is due to the fact that *all* predefined definition modules for arithmetic are in their turn implicitly imported in the definition module delta.

```
DEFINITION MODULE delta;
IMPORT deltaB, deltaC, deltaI, deltaR, deltaS, deltaM;
```

8.5 Unique types and destructive updates

Section 2.6 describes the Miranda method of writing interactive programs. However, this approach has many disadvantages. To interact with the operating system the initial expression has to yield special constructors which have to be interpreted by the run-time system. To ensure referential transparency, certain restrictions are imposed on the kind of I/O that is permitted (e.g. a file cannot be rewritten). Furthermore, the solution is rather inefficient and the structure of highly interactive programs can become rather ugly. Generally, there is a direct relation between input and output which is not always clearly visible in the Miranda approach: input is converted into a list to read from, output has to be produced by an eager printing of the result yielded by the initial expression. Since the order in which functions are evaluated is hard to predict, a frequently occurring error consists of the specification of a program that demands input before a prompt for such is given. So the Miranda solution is not a satisfactory method for writing interactive programs.

On the other hand, it is not easy to find a good solution. The problem is that certain kinds of destructive updates, which occur for example in traditional file I/O, would be very nice to have. But one cannot have destructive updates in the functional framework without some drawbacks. So the questions that arise are:

- How is it possible to perform destructive updates efficiently without violating the functional properties of the program (referential transparency, Church–Rosser property)?
- Since the evaluation of a functional program is unpredictable, how can one control the order in which destructive updates are performed?

When are destructive updates safe?

What kind of problems are caused by functions that perform destructive updates? Such functions are, for example, functions that destructively update an array, instantaneously write to a file on disk or to a window on a screen.

Take, for example, file I/O. The most obvious and efficient way to perform file I/O is by implementing functions that directly read and write to a file, such as is common in imperative languages. However, a naïve implementation of such functions in a functional language would conflict with referential transparency. For instance, assume a function that upon evaluation directly writes a character to a given file. Assume that such a function is of type:

```
:: FWriteC CHAR FILE   ->     FILE;
```

This function takes a character and a file as an argument. However, the character cannot be written into the given file and returned as result because the original file can be shared and used in other function applications. Modification of the argument will therefore also affect the outcome of other computations that share the same argument. The result of a program will now depend on the evaluation order, and the Church–Rosser property is lost.

Example of an illegal destructive update:

```
:: F  FILE ->    (FILE,FILE);
   F  file   ->    (file1, file2),
                   file1:  FWriteC 'a' file,
                   file2:  FWriteC 'b' file;
```

> Assume that the function FWriteC would actually append the given character to the file it gets as an argument. Now, since the file is used twice in the function body of F the result will depend on the evaluation order. It will either be (file++'a', file++"ab") or (file++"ba", file++'b'). But, if FWriteC does not change the original file the result would be (file++'a', file++'b'), independent of the chosen evaluation order.

The destructive update is also in conflict with the standard graph rewriting semantics of FWriteC (see also Appendix B) that prescribes construction of a *new* file (contractum) in the function body with the contents of the given file and the given character. So each time a character is written such a data structure has to be constructed. This is of course very inefficient and it is not the intention either. One really wants to have the possibility of modifying an *existing* file *instantaneously*. The problem becomes even more obvious when one wants to write to a window on a screen: one would like to be able to draw in an *existing* window. In the standard semantics one would be obliged to construct a new window with each drawing command.

However, under certain conditions destructive updates *can* be allowed. If it can be guaranteed that an argument of a particular function application is not used by (shared with) other function applications, then the argument becomes garbage if it is not used in the corresponding function body. In principle one can destructively update such an argument to construct the function result. This would make it possible to construct a function FWriteC that can actually write the character to the given file, yielding the updated file. But, such a function can only restrictively be used. For example, the illegal example above would indeed not be allowed.

Controlling the evaluation order

Assume that we can define functions that perform destructive updates in such a way that the Church–Rosser property of the program is retained. One certainly would not want destructive updates such as interaction with the user to occur in arbitrary order. How can one ensure that destructive updates are performed in the intended order? A well-known method is environment passing. The state of the environment one wants to regard is then coded into an (abstract) object (e.g. a FILE in the case of file I/O). Each function that modifies the environment needs the current state of the environment as argument and yields the updated environment as result. In the case of file I/O this means that all functions that perform file I/O need a file as argument and return an updated file (see Section 8.5.2 and Appendix B). So such an object has to be passed from one function to another. When a function performs an update of an argument, it must be guaranteed that all previous updates of that argument by other functions have already taken place. So a function that updates an argument must be **hyper-strict** in this argument, i.e. it must be guaranteed that the argument is always in normal form before the function is applied. Passing around unique objects from just one function to another is called the **single-threaded** use of objects. For such use destructive updates are allowed and they take place in a fixed sequential order (innermost).

Solutions for incorporating destructive updates

Currently, there are two promising ways to incorporate destructive updates as described above in a pure functional language. The first method is by using **monads** (Peyton Jones and Wadler, 1993). Monads are abstract data structures on which functions are defined in such a way that a single threaded use of the monad is guaranteed. However, a disadvantage of the method is that one can only manipulate one monad at a time. So, for example, destructive file I/O and destructive array manipulation are hard to combine. In Clean a second method is introduced that does not have this disadvantage but it does require a special type system. One can assign a *unique type attribute* (Smetsers *et al.*, 1993) to an arbitrary object that can be used to guarantee that such an object can be destructively updated safely.

8.5.1 The unique type attribute

A node *n* of a graph *G* is **unique** with respect to a node *m* of *G* if *n* is only reachable from the root of *G* via *m* and there exists exactly one path from *m* to *n* (Figure 8.2).

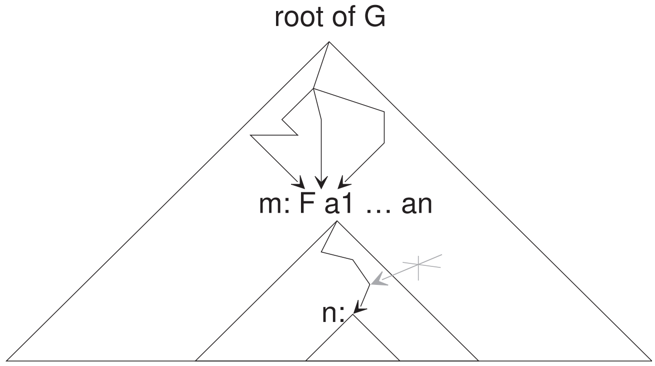

Figure 8.2 Unique node n with respect to the root m of the redex.

A property of a unique node is the fact that it has a reference count (in-grade) of one. A shared node which has a larger reference count than one clearly cannot be unique with respect to any node. A reference count of one is, however, not sufficient for uniqueness with respect to any other node: there can still be more paths leading to such a node. 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.

It would be nice if at compile-time the uniqueness of arguments and results of functions could be determined. Unfortunately, this is undecidable. In Clean a decidable approximation has been incorporated using **unique types**. Unique types (Smetsers *et al.*, 1993), defined on *graphs*, have many similarities with linear types, defined on *λ-terms* (Girard, 1987; Wadler, 1990). An important difference is that Clean's unique types give information about the way *a specific function* has to be *applied* (e.g. this function has to be called with an argument that is used in a linear way) while other linear type systems give information about the way expressions are being used in the function body (e.g. this argument of the function is used in a linear way in the function body).

Unique type specifications and functions

The type of a graph in a rewrite rule can have the **unique type attribute**, i.e. the graph can be of type `UNQ T`. *If a graph on a left-hand side or on the right-hand side of a rewrite rule is of type* `UNQ T`*, it is guaranteed that at run-time the root of the corresponding graph is unique with respect to the root of, respectively, the function application or contractum. When no contractum is specified, a result of type* `UNQ T` *means that a redirection is performed to an object that was unique with respect to the root of the function application.* When this cannot lead to confusion the phrase 'a unique graph' will be used instead of 'a graph which is unique with respect to the function application/contractum'.

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

- 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. The type specification of the function `F` defined below will be rejected since two paths are leading from the root of the contractum to the same node x. The unique argument of the function F is not unique any more in the resulting pair. So the following types for F would be accepted: ::F UNQ INT -> ([INT], [INT]) or ::F INT -> ([INT], [INT]).

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

Each function alternative is separately checked for consistency. Although in the definition of F below the unique node-id n is used twice this does not lead to a conflict: only one of the guarded expressions is chosen at run-time.

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

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

It is not always wrong to use a 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.* The reason is that a guard is evaluated in advance to determine which of the function alternatives has to be chosen. A guard will yield a Boolean result that cannot contain references to UNQ nodes. To guarantee that no destructive updates occur when a guard is evaluated, a graph that is being used in both guard and guarded expression will lose its unique type attribute in the guard. This enables the non-destructive (observing) inspection of such a graph in a guard. This property for guards can be generalized to a more general property for uniqueness connected to the order of evaluation (see Smetsers *et al.*, 1993).

Example of sharing of a unique node-id in guard and right-hand side:

```
:: 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 system will assign the type INT (and not UNQ INT) to both uses of n in the guard.

- Demanded UNQs have to be obeyed (with an exception for function types, see below): 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 attribute occurs, there has to be a corresponding position in T' with the UNQ attribute.

Not well-typed example. F is recursively called with a graph of type INT, while in the type specification of F an argument of type UNQ INT is demanded. This will yield a type error.

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

:: G  INT       ->    INT;
   …
```

For demanded *function types* with UNQ arguments the offered argument type need not be unique: a demanded type => UNQ T T' and an offered type => S S' are unifiable (when in the standard sense, ignoring the UNQ attributes, => T T' and => S S' are unifiable).

- When a UNQ type is offered but not demanded, this offered type is accepted (with an exception for function types, see below). This can be explained by the fact that a UNQ type imposes an additional restriction on the way the object is used. If this restriction is not demanded by the context this is of course fine. It does mean that *type conversions* have to take place. An offered object of type UNQ T is automatically converted to type T if this type is demanded from the context.

Take the previous example and consider the application of G n. G demands an argument of type INT while G is applied with a graph of type UNQ INT. The type system will automatically convert the UNQ INT to an INT and will then conclude that the type of the application G n is correct.

Offered *unique function types* are never converted: when a type UNQ => T T' is offered, the demanded type must be of type UNQ => S S' and unifiable in the standard sense.

```
:: F UNQ FILE CHAR      ->    UNQ FILE;
   F f c                ->    FWriteC c f;

:: G (=> CHAR UNQ FILE)  ->    (UNQ FILE, UNQ FILE);
   G g                   ->    (g ‘a’, g ‘b’);
```

With the well-typed rules above the expression G (F StdErr) would be rejected. The type of F StdErr is UNQ (=> CHAR UNQ FILE) owing to the left to right UNQ propagation rule which is explained in the next subsection.

Furthermore, offered *function types* with UNQ arguments have to be obeyed: when a demanded type => T T' and an offered type => UNQ S S' are unifiable (in the standard sense, ignoring the UNQ attributes) then they are not unifiable taking UNQ into account. The demanded type must be => UNQ T T'.

```
:: UMap (=> UNQ x UNQ y)   [UNQ x]     ->   [UNQ y];
   UMap f                  [ ]         ->   [ ];
   UMap f                  [ x | xs ]  ->   [ f x | UMap f xs ];

:: WriteAll CHAR   [UNQ FILE]   ->   [UNQ FILE];
   WriteAll c      ufiles       ->   UMap (FWriteC 'a') 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 offered 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. Note that there is no distinction between the definition of the function Map and UMap: there is only a difference in the specified type.

When functions are allowed to be polymorphic in their UNQ attributes one can define a generic Map function that can be used both as the standard Map function and as the UMap function in the example above.

Defining types with UNQ attributes

A programmer can add unique type attributes to any type. List, tuples and function types, as well as their subtypes, can have the unique attribute. Furthermore, 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.

An important aspect of a function application containing a unique graph is that it cannot be a part of a graph that is not unique with respect to that function application. In other words, 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 any more. The propagation rule holds for all types, and also for function types. For non-function types UNQ attributes propagate from **inside-out**, for function types it propagates from **left to right**. In Clean, it is not necessary to give the full type specification; the propagation of the UNQ attributes is done automatically by the compiler.

Example of '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).
Example of 'left to right' propagation: take F typed 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 F 3 has to be unique as well. Hence, the full type specification for F is => UNQ INT UNQ (=> INT INT).

Synonym types

Synonym types can be used in the usual way.

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

Applying UNQ propagation to achieve the full type specification for this example gives UNQ UTUPLE UNQ x -> UNQ (UNQ x, UNQ x).

Algebraic types

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

(1) All occurrences in both left- and right-hand sides of a certain type variable must have the same type attribute.
(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.

A list with a unique spine but not necessarily with unique elements:

```
::  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 full definition deduced by the compiler is:

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

For clarity, we advise specifying the left-hand side completely:

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

A list with unique elements and hence with a unique spine:

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

Owing 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 become unique and therefore so does 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 parametrized with a unique type is rather obvious: the resulting type is deduced from the type obtained by substituting the parameters (including the UNQ attributes) in the definition of T.

Take for example the following common definition of a list:

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

Then for the type List UNQ INT the following full type definition is deduced:

```
:: 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));
```

Left to right propagation deduces for the type T UNQ INT the following:

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

Abstract types

Abstract types with UNQ attributes are specified by giving the *full* type specification of the left-hand side of the corresponding type definition in the implementation module.

The previous fully unique List type is turned into an abstract type as follows:

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

8.6 Input/output handling

Using the UNQ type attribute, single-threaded use of *file I/O* and *screen I/O* (in Clean called *event I/O*) can be assured. In this way incremental updates of persistent data are made possible (see Section 8.6.2). Specification of modern I/O that uses modules, windows, menus and the like can be done on a very high level of abstraction by making use of the predefined Clean library (Achten *et al.*, 1993) that heavily uses UNQ types. This library is part of Appendix B.

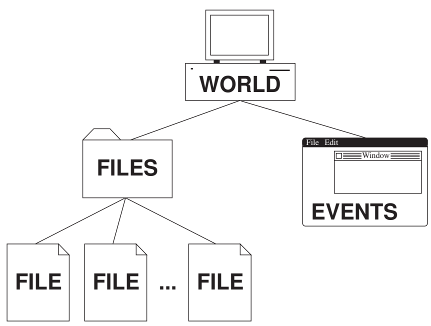

Figure 8.3 The I/O environment hierarchy of Clean.

The I/O constructs of Clean are referentially transparent and hence '100% functional'. An abstract object representing the outside world has been predefined with type UNQ WORLD. From this world the following disjoint abstract objects can be extracted (see Figure 8.3): an object of type UNQ EVENTS which is used to perform screen I/O and an object of type UNQ FILES (the file system) from which in its turn objects of type FILE (concrete files) can be extracted (opened) to perform file I/O. One can open unique files (if these files have to be modified) or non-unique ones (for read-only files).

8.6.1 The world

The world contains all the information about the concrete environment that is relevant to the program. There is no function to create a world. The world is an abstract object of predefined type UNQ WORLD optionally given to a Clean program as an argument of the Start rule:

```
RULE
:: Start UNQ WORLD -> ...    ==   any type
   Start w -> ...            ==   any computation yielding the indicated type;
```

Pure computations ignore the world and its subenvironments. Interactive programs need to access and change the world using functions which require a unique world. When the world becomes garbage it does not mean that the world has ceased to exist, but it means that the program no longer performs operations on the world. If the subenvironments have been retrieved earlier, then they can still be accessed in the program.

8.6.2 File I/O

FILES is the unique subenvironment of the world containing all the files that are visible to the program (the file system). The file system is retrieved from a unique world by the rule OpenFiles and can be put back again by the rule CloseFiles. Once the file system has been retrieved from the world, it cannot be retrieved again without closing it first.

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

A Clean file has type FILE. To open a file (to read or write) one needs the file system. Only writable files are opened as UNQ FILE; read-only files do not require the unique attribute. For example, the following functions have been predefined (there are many more: see Appendix B).

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

In the Clean library also an FSeek function is predefined with which the file pointer can be moved explicitly such that a file can be randomly accessed. Note that all functions return a (UNQ) FILE on which I/O functions can continue.

The following example illustrates the use of WORLD, FILES and FILEs:

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

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

This program copies the contents of one file to another. First it retrieves the file system from the world. This file system is used to open the source and the destination file. The world and the file system are no further needed and become garbage. The source file is only being read (indicated by FReadData), so it does not have to be unique. The destination file is being written (FWriteData) and therefore this file must be unique. After completion of copying, the source file becomes garbage, and the program yields the written file. To do the actual copying on the open files CopyF is applied. When the source file is empty the destination file is yielded, otherwise CopyChars reads a character from the source file, writes it to the destination file and continues recursively.

It is possible that a UNQ FILE is used in such a way that it loses its UNQ attribute (e.g. when the file gets shared). Since all destructive operations on files require an object of type UNQ FILE, an ordinary (possibly shared) file of type FILE cannot be modified any more. But there are several non-destructive operations defined on an object of type FILE with which such a file can still be examined.

8.6.3 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 are event driven: actions performed by the user generate events to the program. Clean's event I/O library (written in Clean) offers a way to define these graphical interface objects on a level that is very close to the way these objects actually appear on the screen. The library completely takes care of all the low-level event handling.

Devices

A **device** in Clean is an interface object such as a window or a menu. There are four devices predefined in the Clean library: the *menu device*, the *window device*, the *dialog device* and the *timer device*.

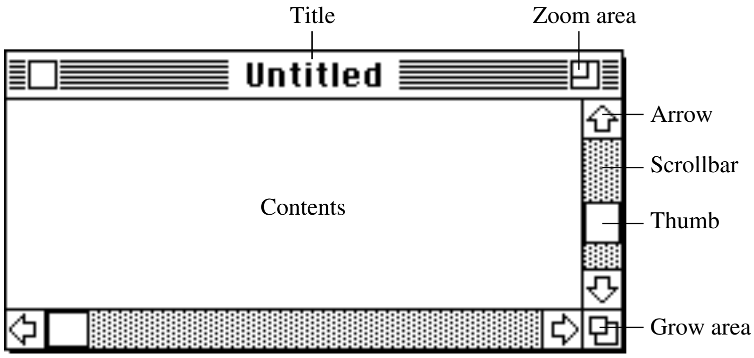

Figure 8.4 Example of a window.

A **window device** (see Figure 8.4) is an interactive device: it reacts on key presses and releases and mouse clicks and releases, coupled with mouse positions. Windows are the only way a program can visualize output. A window gives a view on a **picture** (again a UNQ abstract object) on which a set of drawing functions is defined.

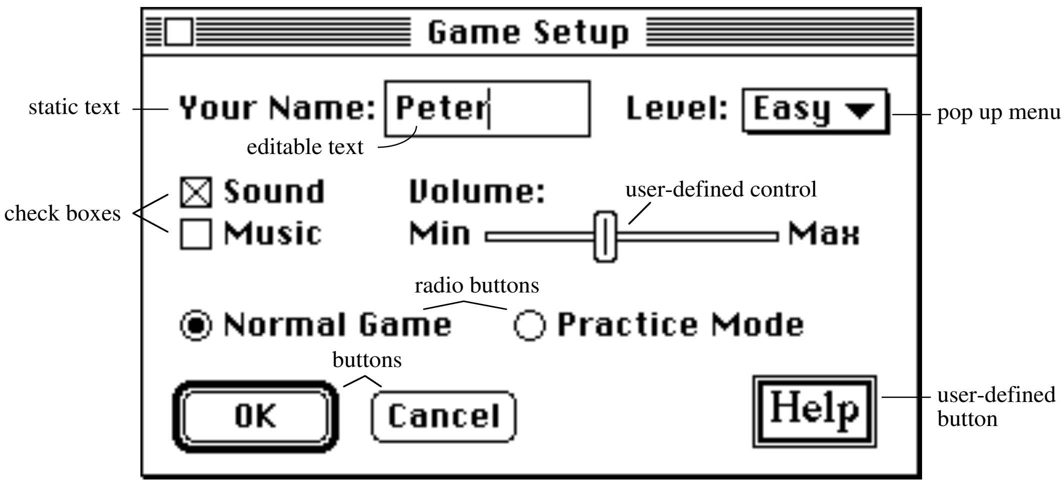

Figure 8.5 Example of a dialog.

The relationship between the visible part of the picture and the window is defined by the positions of the thumbs of the window's scroll bars. One can open several windows dynamically, at most one of them is active. Almost all window management is handled automatically by the library. The programmer just has to define the update management which is the redrawing that should be performed if an invisible part of the window becomes visible again. The programmer can also define what should be done when a window is closed.

The **menu device** (see Figure 8.7) conceptualizes choosing from a distinguished set of available commands. A menu device can contain pull-down menus each containing commands or submenus.

The **dialog device** (see Figure 8.5) conceptualizes structured communication between the program and the user via a form that has to be filled in. One can have modal and modeless dialogs, as well as notices. Modal and modeless dialogs can contain editable text fields, static text fields, pop-up menus, radio buttons, check boxes, buttons, final buttons and user-defined controls.

With the **timer device** a program can be synchronized (see Section 8.6.4): a function can be evaluated every time a certain time interval has passed. Several timers can be installed. When a time interval is set to zero, a timer event is generated whenever no other event is generated.

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. 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. The program state can be used by the programmer to maintain the current state of the program (which is dependent on the kind of program).

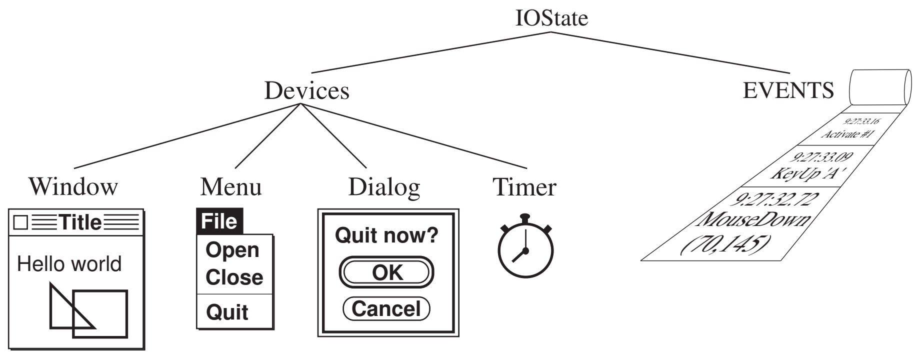

Figure 8.6 The contents of the IOState.

The **I/O state** is an abstract data type of type IOState that can be regarded as a *hidden* state used by the library to maintain the current state of the I/O interface. All interactions and I/O functions are defined on this 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 I/O state contains (see Figure 8.6):

- the event stream (modelled by the EVENTS);
- the definition of all the devices that participate in the interaction.

Transitions are triggered by the events in the event stream. Each event in the event stream is dispatched in the library to the proper device, which computes the next interaction state by calling the proper event handler.

The event stream

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

The definition of the devices

Devices are specified with the help of a predefined algebraic data type DeviceSystem. With this type actually a special kind of declarative language is introduced in which the programmer can specify the properties of the devices that are being used (see the example in Section 8.6.4). The programmer has to specify:

- *which* of the predefined I/O devices are being used;
- *how* these devices should be initialized;
- *which* **event handler** or **call-back routine** *has to be called* when a certain active device has been triggered (e.g. a particular menu item that has been chosen). An event handler is a user-defined higher order state transition function that takes the program state and I/O state as an argument to yield a pair with the new program state and the new I/O state as result.

Starting and terminating interactions

Starting and terminating interactions are handled by two special functions: StartIO and QuitIO. StartIO takes the specification of the set-up of the I/O system as described above, the initial program state s_0, a possibly empty list of functions that can be used to change the default settings of the system, and the event stream obtained from the world. StartIO takes the description of the devices, after which these devices are activated and drawn on the screen. Then, the device description and the event queue are stored in the I/O state. Finally, the default settings are set and an internal function DoIO is called with the initial program state

s_0 and the initial I/O state $IOState_0$. This function DoIO recursively calls itself. In each call an input event e_n is taken from the event stream. The input event is dispatched to the proper device which computes the next program state s_{n+1} and the next I/O state $IOState_{n+1}$ by applying the proper event handler. In this way a sequence of pairs of program state and IOState, starting from the program state s_0 and I/O state $IOState_0$, is computed. Below the simplified implementation of StartIO is given using internal library functions InitIO, DoIO, GetEvent, GetEvents and GetHandler.

```
StartIO device_defs program_state default_settings events
->     DoIO (program_state, io_state),
       io_state:    InitIO device_defs default_settings events;

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

The order of evaluation guarantees that the transition triggered by event e_{n+1} is only reduced after the transition triggered by e_n has yielded a complete $IOState_{n+1}$. The interaction obtained in this way can only be terminated by having any of the device functions apply QuitIO to its IO-State argument. The function QuitIO produces a special I/O state, Closed-IO_State, in which all devices are closed. DoIO matches on this special state producing the final program state and the remaining event stream.

8.6.4 An example: the game of Life

In this section we present an example of a typical interactive Clean program that uses the Clean I/O library. The program describes the interface for a system playing the game of Life. This is a ‘game’ consisting of an infinite two-dimensional space (the universe). A cell is identified by a Cartesian position in the universe. A cell is either alive or dead. When an initial generation of living cells is given, each following generation is computed as follows:

- if a living cell has just two or three living neighbour cells, it survives to the next generation;
- if a dead cell has exactly three living neighbour cells, it becomes alive in the next generation.

In this example we concentrate on the specification of the interactive part of the program, so our prime interest is not, for instance, the function that calculates a new generation given the current generation.

Hence, we assume that the specifications of such functions are already given.

The following functions are assumed to be given: LifeGame computes, given the current generation of cells, a triplet consisting of the next generation, and, to make drawing of cells easier, the collection of new-born cells and the cells that have passed away. RemoveCell deletes a cell from a generation; adding a cell to a generation is done by AddCell. They are used for the creation of the initial generation by the user.

```
:: LifeGame        Generation     ->   (Generation, Generation, Generation);
:: RemoveCell      LifeCell Generation   ->   Generation;
:: AddCell         LifeCell Generation   ->   Generation;
:: InitialGeneration    ->    Generation;
```

The initial generation of cells has to be specified by the player. The player can add or delete a cell at a certain position by clicking with a *mouse* in a *window*. Then the calculation of generations can be started. There are several options that can be chosen from a *menu* (Figure 8.7).

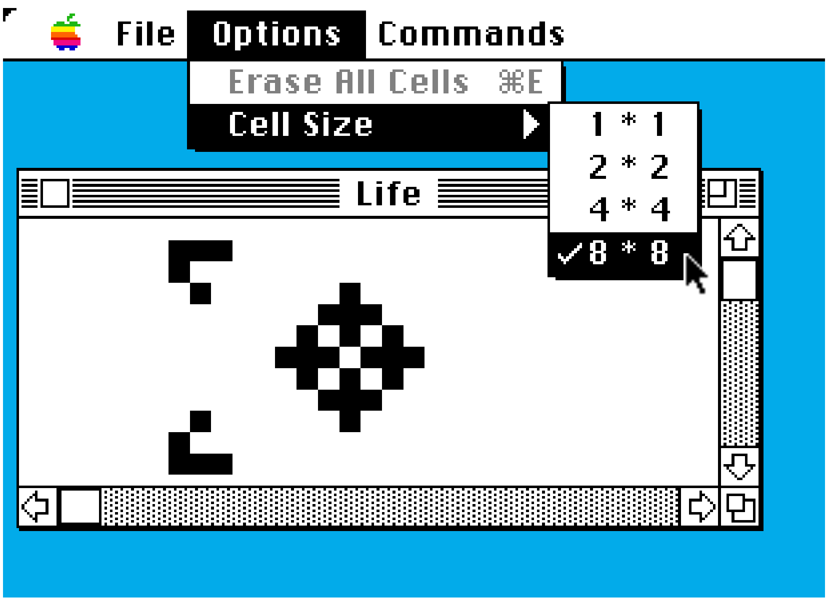

Figure 8.7 **Life** window, **Options** menu and **Cell Size** submenu.

The menu consists of:

- a pull-down menu named **File**, containing:
  – a **Quit** command to terminate the application.
- a pull-down menu named **Options**, containing:
  – an **Erase** command to clear the contents of the window and to create a fresh game with an empty generation;

  - a submenu called **Cell Size** to change the size of the cells displayed in the window. Because at most one size is valid at a time, the items are organized as radio items (selection of an item causes the deselection of all others). The initial cell size will be set to **8 * 8** pixels.
- a pull-down menu named **Commands**, containing:
  - a **Play** command to start the continuous computation of next generations. **Play** disables itself such that the command cannot be chosen again. Furthermore, **Erase** will be disabled and **Halt** has to be enabled.
  - a **Halt** command to terminate the computation of generations and reverse all (dis)abling done by **Play**.

When **Play** is chosen from the menu, generations will be computed one after another. If nothing happens, this computation will go on forever. To stop it one has to choose **Halt** from the menu. To make this possible one has to ensure that the program inspects its event queue regularly to see if, for example, **Halt** has been chosen. This can be realized by using the *timer* device with the time interval set to zero. When no events are generated by the user, a timer event is generated automatically that is used to calculate the next generation. If there is a user event, it will get priority. In this way the calculation of generations can be interrupted.

```
MODULE LifeGame;
IMPORT delta;
IMPORT deltaEventIO, deltaMenu, deltaWindow, deltaTimer, deltaPicture;
IMPORT Life;
```

Constants to enhance comprehension and maintenance:

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

The program state State consists of the current generation and the size of the cells displayed in the window. A synonym for the IOState is defined to prevent type rules from cluttering up.

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

The execution of the program starts here. A window is created, to access a null device (which is not active yet) and a menu. The parameters have to obey the algebraic data type defined in the library. The symbols printed in *italic* are the user-defined event handlers. To each menu item a different label (an arbitrary constant number) is attached (an Id) such that one can refer to it later. Observe the close relation between the definitions of the window and menu device and their graphical appearance on the screen.

```
RULE
:: Start UNQ WORLD -> (State, UNQ EVENTS);
   Start world    ->
   StartIO [ menus, timer, window ] initial_program_state [ ] events,
   initial_program_state:  (InitialGeneration, 8),
   (events, world'):     OpenEvents world,
   menus:     MenuSystem [ file, options, cmnds ],
   file:      PullDownMenu FileMenuId "File" Able
               [  MenuItem QuitId "Quit" (Key 'Q') Able Quit    ],
   options:   PullDownMenu OptionsMenuId "Options" Able
              [   MenuItem EraseId "Erase All Cells" (Key 'E') Able Erase,
                  SubMenuItem CellSizeId "Cell Size" Able
              [   MenuRadioItems Size8Id
              [   MenuRadioItem Size1Id "1*1" NoKey Able (ChangeSize 1),
                  MenuRadioItem Size2Id "2*2" NoKey Able (ChangeSize 2),
                  MenuRadioItem Size4Id "4*4" NoKey Able (ChangeSize 4),
                  MenuRadioItem Size8Id "8*8" NoKey Able (ChangeSize 8)
              ]   ]   ],
   cmnds:     PullDownMenu CommandsMenuId "Commands" Able
              [   MenuItem PlayId "Play" (Key 'P') Able Play,
                  MenuItem HaltId "Halt" (Key 'H') Unable Halt
              ],
   timer:     TimerSystem
              [   Timer TimerID Unable NullInterval NextGeneration  ],
   window:    WindowSystem
              [   ScrollWindow LifeWindowId WindowPos "Life"
                  ScrollBarH&V ScrollBarH&V PictureRange
                  MinimumSizeOfWindow InitialSizeOfWindow
                  UpdateWindow [ Mouse Able Track ]     ];
```

Next we define the user-defined event handlers. All handlers take two additional arguments of type State and IO and yield a pair of type (State, IO).

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

:: Erase State IO -> (State, IO);
   Erase (g, s) io
   ->     (([ ], s), DrawInActiveWindow [ EraseRectangle PictureRange ] io);

:: ChangeSize INT State IO -> (State, IO);
   ChangeSize ns (g, s) io
   ->     ((g, ns), DrawInActiveWindow cells io),
          cells:     [ EraseRectangle PictureRange | Map (DrawCell ns) g ];

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

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

A timer event is generated when no other event is present and this is used to calculate a new generation of cells:

```
:: NextGeneration TimerState State IO -> (State, IO);
   NextGeneration tstate (g, s) io
   -> ((ng, s), ChangeIOState [     DrawInActiveWindow erasecells,
                                    DrawInActiveWindow newcells  ]    io),
      erasecells:       Map (EraseCell s) died,
      newcells:         Map (DrawCell s) new,
      (ng, new, died):  LifeGame g;
```

UpdateWindow redraws all cells regardless of their visibility:

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

Track evaluates all mouse activities in the window:

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

Auxiliary drawing functions have to be defined. DrawCell draws the cells that have been created and EraseCells draws the cells that have died.

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

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

Note that the IOState (the parameter of type IO) is used single-threadedly everywhere in the program (otherwise this program would have been rejected by the compiler). In the example above this is generally done implicitly (using the predefined function ChangeIOState). It can also be done explicitly. Take for example the Play function as defined above. This function can also be defined as follows:

```
:: Play State IO  ->    (State, IO);
   Play state io  ->    (state, play),
                        play :    EnableTimer TimerID
                                      ( EnableMenuItems [ HaltId ]
                                         ( DisableMenuItems [ PlayId, EraseId ]
                                            ( DisableActiveMouse io ) ) );
```

Or even more explicitly:

```
:: Play State IO  ->    (State, IO);
   Play state io  ->    (state, io4),
                        io1:   DisableActiveMouse io,
                        io2:   DisableMenuItems [ PlayId, EraseId ] io1,
                        io3:   EnableMenuItems [ HaltId ] io2,
                        io4:   EnableTimer TimerID io3;
```

Summary

- Clean is a *lazy*, *higher order functional* programming language based on FGRSs.
- Clean can also be used as *intermediate language* in the compilation path from functional languages to concrete machine architectures.
- *Sharing of computation* can be expressed explicitly by the programmer in a general way, including the possibility of specifying cyclic structures.
- Clean is a *strongly typed* language (based on Milner–Mycroft typing extended for the use of patterns) including *polymorphic types*, *abstract types*, *algebraic types* and *synonym types*, as well as basic types; there are predefined type constructors for *lists*, *tuples* and (curried) *functions*. The type system enhances efficiency, modularity and understandability.
- A Clean program can be decorated with *annotations* that influence the evaluation order.
- With *strict annotations* the evaluation can be made (partially) *eager* instead of *lazy*. Strict annotations are only safe when they are used in such a way that the termination behaviour of the program is not changed. Generally, the annotations are placed on the strict arguments of a function.
- Annotations can be defined by the *programmer* but they can also be generated automatically by the built-in *strictness analyser* of the Clean system. An annotated program generally runs much faster and has a much better space behaviour than a program without annotations.
- Clean has a *modular structure* with *implementation modules* and *definition modules* including a facility to import definitions from other modules implicitly and explicitly; it includes predefined libraries for basic operations (δ-rules) on objects of basic types.
- A *unique type attribute* can be added to the type specification of any object. The type system ensures that such an object can only be used *single-threadedly*. When such an object is passed to a function it can in principle be reused for the construction of the function result. In this way functions can be constructed that perform destructive updates without loosing referential transparency.
- With the help of the unique type attribute and higher order functions a library has been written in Clean that enables a *high-level specification* of *interactive programs* that use *windows*, *dialogs*, *menus* and the like.

EXERCISES

8.1 Describe *all* reduction paths for the initial expression G (H 0) given the rules below if they are valid (a) Miranda, (b) Clean, (c) TRS, (d) GRS rules. Ignore small syntactic differences between the systems.

```
F (H 0)   x = x
F 1       x = x
H x = 1
G x = F x x
```

In the following exercises a task is given for which a Clean program has to be written. Specify the type of each rule. Invent appropriate (partially strict) data structures when necessary. Utilize sharing and cycles to prevent computations being carried out more than once.

8.2 Compute a list of all Fibonacci numbers.

8.3 Compute the powerset of a given set (represent a set as a list, multiple occurrences of an element in the list are not allowed).

8.4 Given a function f and a set of values X, compute the closure of f and X: $clos(f, X) = \{ f^n(x) \mid n \in Nat, x \in X \}$.

8.5 Compute Pascal's triangle (to a certain depth) in a neatly formatted way. This triangle consists of numbers such that each number is the sum of the numbers just to the right and just to the left of it, on the line above it. The sides of the triangle consist of 1s. Part of the triangle is given here:

```
         1
      1     1
   1     2     1
1     3     3     1
```

8.6 Read a file, named 'perm.in', containing some permutation of the numbers 1 to n. To determine n ask the user for input. After reading in this file find out which permutation cycles the permutation in the file consists of. Write these permutation cycles to a file named 'cycles.out'.

Suppose the file contains 2 4 3 1, which is a permutation of the numbers 1 to 4; then it consists of the permutation cycles (1 2 4) and (3).

8.7 Implement the functions LifeGame, RemoveCell, AddCell and InitialGeneration as applied in Section 8.6.4.

8.8* Refine the CopyF function of Section 8.6.2 such that it asks interactively the names of the files. Open StdIO to get a standard window to prompt the user for the file names. Open StdErr to display error messages. Allow StdIO as one of the file names. Make the program robust such that wrong file names cannot lead to abortion of the program. Make a more user-friendly version of the program by using file selector dialogs.

8.9* Program a simple pocket calculator that can be used by pressing buttons in a dialog definition.

8.10* Write the well-known Tetris game. Define suitable menus and dialogs. Add a help file and a high-score list.

8.11* Write a matrix multiplication program using a UNQ ARRAY INT and using an [[INT]]. Measure the time–space behaviour of both representations and explain the differences.

8.12* Write in C an imperatively written program (assignments allowed) that sorts a list using the quicksort algorithm. Write a quicksort function in Clean. Also write the quicksort algorithm defined on lists with a unique spine. Measure the differences in time–space behaviour and explain them.

8.13* Write a non-trivial program (e.g. the λ-reducer of Exercise 3.7) in Clean, an imperative language, an object oriented language and a logical language. Which style of programming did you like most? Why? Measure and compare the efficiency in both time and space. Also measure the effect of the strictness analyser.