Version 3.0 Language Report

Preface

CLEAN is a practical applicable general-purpose lazy pure functional programming language suited for the development of real world applications.

CLEAN has many features among which some very special ones.

Functional languages are usually implemented using graph-rewriting techniques. CLEAN has explicit graph rewriting *semantics*; one can explicitly define the *sharing* of *structures* (*cyclic structures* as well) in the language (Barendregt et al., 1987; Sleep et al., 1993, Eekelen et al., 1997). This provides a better framework for controlling the time space behavior of functional programs.

Of particular importance for practical use is CLEAN's Uniqueness Type System (Barendsen and Smetsers, 1993a) enabling the incorporation of destructive updates of arbitrary objects within a pure functional framework and the creation of direct interfaces with the outside world.

CLEAN's "unique" features have made it possible to predefine (in CLEAN) a sophisticated and efficient I/O library (Achten and Plasmeijer, 1992 & 1995). The CLEAN Object I/O library enables a CLEAN programmer to *specify interactive window based I/O applications* on a very high level of abstraction. One can define callback functions and I/O components with arbitrary local states thus providing an object-oriented style of programming (Achten, 1996; Achten and Plasmeijer, 1997). The library forms a platform independent interface to window-based systems: one can port window based I/O ap-plications written in CLEAN to different platforms (we support Mac and PC) without modification of source code.

Although CLEAN is *by default* a *lazy language* one can smoothly turn it into a *strict language* to obtain optimal time/space behavior: *functions* can be defined *lazy* as well as *(partially) strict* in their arguments; any (recursive) *data structure* can be defined *lazy* as well as *(partially) strict* in any of its arguments.

The rich type system of CLEAN 1.3 (offering high-order types, polymorph types, type classes, uniqueness types, existentially quantified types, algebraic types, abstract types, synonym types, record types, arrays, lists) is extended with *multi parameter type constructor classes* and *universally quantified types* currently limited to rank 2, rank n is in preparation). Furthermore, arrays and lists are better integrated in the language. Strict, spine-strict, unboxed and overloaded lists are predefined in the language.

CLEAN now offers a hybrid type system with both static and dynamic typing. An object (expression) of static type can be changed into an object of dynamic type (a "`Dynamic`") and backwards. One can read a `Dynamic` written by another CLEAN program with one function call. A `Dynamic` can contain data as well as (unevaluated) functions. This means that one can very easy transfer data as well as code (!) from one CLEAN application to another in a *type safe* manner enabling *mobile code* and *persistent storage* of an expression. This technique involves just-in-time code generation, dynamic linking and dynamic type unification.

CLEAN offers support for generic programming using an extension of the class overloading mechanism. One can define functions like equality, `map`, `foldr` and the like in a generic way such that these functions are available for any (user defined) data structure. The generic functions are very flexible since they not only work on types of kind star but also on higher order kinds.

CLEAN (Brus *et al.*, 1987; Nöcker *et al.*, 1991; Plasmeijer and Van Eekelen, 1993) is not only well known for its many features but also for its fast compiler producing very efficient code (Smetsers *et al.*, 1991). The new CLEAN 2.0 compiler is written in CLEAN . The CLEAN compiler is one of the fastest in the world and it produces very good code. For example, the compiler can compile itself from scratch within a minute.

The CLEAN 2.0 system includes lots of tools and libraries, all written in CLEAN of course. Included is an IDE (Integrated Development Environment), a dedicated text editor, a project manager, a code generator generating native code (the only piece of software written in C), a static linker, a dynamic linker, a proof system (Sparkle), a test system (GAST), a heap profiler, a time profiler, and lots of libraries.

People already familiar with other functional programming languages (such as Haskell; (Hudak et al., 1992), Gofer/Hugs (Jones, 1993), Miranda (Turner, 1985) and SML (Harper et al., 1986)) will have no difficulty to program in CLEAN. We hope that you will enjoy CLEAN's rich collection of features, CLEAN's compilation speed and the quality of the produced code (we generate native code for all platforms we support). CLEAN runs on a PC (Windows 2000, '98, '95, WindowsNT). There are also versions running on the Mac and Linux.

Research on CLEAN started in 1984 (the Dutch Parallel Machine Project) in which we had to good idea to focuss on compilation techniques for classical computer architectures. Many new concepts came out of the research of the CLEAN team (see below). These ideas are not only incorporated in our own system, many of them have also been adopted by other languages like Haskell and Mercury.

More Information on Clean

A tutorial teaching how to program in CLEAN can be found on our web pages.
See http://wiki.clean.cs.ru.nl/Functional_Programming_in_Clean.

Information about the libraries (including the I/O library) that are available for CLEAN can also be found on the web, surf to http://wiki.clean.cs.ru.nl/Libraries.

There is a manual teaching the use of the Object I/O library. It includes many examples showing you how to write interactive window based programs.
See http://clean.cs.ru.nl/download/supported/ObjectIO.1.2/doc/tutorial.pdf.

The basic concepts behind CLEAN (albeit of one of the very first versions, namely CLEAN 0.8) as well as an explanation of the basic implementation techniques used can be found in Plasmeijer and Van Eekelen (Adisson-Wesley, 1993). The book is out of print, but copies can found on
http://wiki.clean.cs.ru.nl/Functional_Programming_and_Parallel_Graph_Rewriting

There are many papers on the concepts introduced by the CLEAN group (such as *term graph rewriting* (Barendregt *et al*., 1987), *lazy copying* (van Eekelen *et al*., 1991), *abstract reduction* (Nöcker, 1993), *uniqueness typing* (Barendsen and Smetsers, 1993, 1996), CLEAN's *I/O concept* (Achten, 1996 & 1997), *Lazy Copying* for Concurrent CLEAN (Kesseler, 1991 & 1996), Type dependent Functions for Dynamics (Pil, 1999), I/O of Dynamics (Vervoort, 2001), a Typed Operating System (van Weelden, 2001). For the most recent information on papers (http://wiki.clean.cs.ru.nl/Publications) and general information about CLEAN (http://clean.cs.ru.nl) please check our web pages.

About this Language Report

In this report the syntax and semantics of CLEAN version 2.0 are explained. We always give a motivation *why* we have included a certain feature. Although the report is not intended as introduction into the language, we did our best to make it as readable as possible. Nevertheless, one sometimes has to work through several sections spread all over the report. We have included links where possible to support browsing through the manual.

At several places in this report context free syntax fragments of CLEAN are given. We sometimes repeat fragments that are also given elsewhere just to make the description clearer (e.g. in the uniqueness typing chapter we repeat parts of the syntax for the classical types). We hope that this is not confusing. The complete collection of context free grammar rules is summarized in Appendix A.

Some Remarks on the Clean Syntax

The syntax of CLEAN is similar to the one used in most other modern functional languages. However, there are a couple of small syntactic differences we want to point out here for people who don?t like to read language reports.

In CLEAN the arity of a function is reflected in its type. When a function is defined its uncurried type is specified! To avoid any confusion we want to explicitly state here that in CLEAN there is no restriction whatsoever on the curried use of functions. However, we don't feel a need to express this in every type. Actually, the way we express types of functions more clearly reflects the way curried functions are internally treated.

E.g., the standard map function (arity 2) is specified in CLEAN as follows:

```
map::(a -> b) [a] -> [b]
map f []    = []
map f [x:xs]  = [f x:map f xs]
```

Each predefined structure such as a list, a tuple, a record or array has its own kind of brackets: lazy lists are always denotated with square brackets `[...]`, strict lists are denotated by `[! ...]`, spine strict lists by `[... !]`, overloaded lists by `[|...]] , unboxed lists by [#...]`. For tuples the usual parentheses are used `(...,...)`, curly braces are used for records (indexed by field name) as well as for arrays (indexed by number).

In types funny symbols can appear like `.`, `u:`, `*, !` which can be ignored and left out if one is not interested in uniqueness typing or strictness.

There are only a few keywords in CLEAN leading to a heavily overloaded use of `:` and `=` symbols:

```
function::argstype -> restype            // type specification of a function

function pattern
| guard = rhs                            // definition of a function

selector = graph                         // definition of a constant/CAF/graph

function args :== rhs                    // definition of a macro

::Type args = typedef                    // an algebraic data type definition
::Type args :== typedef                  // a type synonym definition
::Type args                              // an abstract type definition
```

As is common in modern functional languages, there is a layout rule in CLEAN (see 2.3). For reasons of portability it is assumed that a tab space is set to 4 white spaces and that a non-proportional font is used.

Function definition in CLEAN making use of the layout rule.

```
primes:: [Int]
primes = sieve [2..]
where
  sieve:: [Int] -> [Int]
  sieve [pr:r]  = [pr:sieve (filter pr r)]

  filter:: Int [Int] -> [Int]
  filter pr [n:r]
  | n mod pr == 0  = filter pr r
  | otherwise    = [n:filter pr r]
```

Notational Conventions Used in this Report

The following *notational conventions* are used in this report. Text is printed in Microsoft Sans Serif 9pts,

the context free syntax descriptions are given in Microsoft Sans Serif 9pts,

```
examples of CLEAN programs are given in Courier New 9pts,
```

- Semantic restrictions are always given in a bulleted list-of-points. When these restrictions are not obeyed they will almost always result in a compile-time error. In very few cases the restrictions can only be detected at run-time (array index out-of-range, partial function called outside the domain).

The following notational conventions are used in the context-free syntax descriptions:

[notion]	means that the presence of notion is optional
{notion}	means that notion can occur zero or more times
{notion}+	means that notion occurs at least once
{notion}-*list*	means one or more occurrences of notion separated by commas
`terminals`	are printed in **`9 pts courier bold brown`**
`keywords`	are printed in **`9 pts courier bold red`**
`terminals`	that can be left out in layout mode are printed in **`9 pts courier bold blue`**
{notion}/ str	means the longest expression not containing the string str

All CLEAN examples given in this report assume that the layout dependent mode has been chosen which means that redundant semi-colons and curly braces are left out (see 2.3.3).

How to Obtain Clean

CLEAN and the INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) can be used free of charge. They can be obtained

- via World Wide Web (http://clean.cs.ru.nl) or
- via ftp (ftp://ftp.cs.ru.nl in directory *pub/Clean)*

Current State of the Clean System

`Release 2.1 (November 2002).`

Compared with the previous version the following changes have been made.

- Experimental features of the previous version, such as dynamics (see Chapter 8), generics (see Chapter 7) and strict-lists (see 4.2) have been improved and further incorporated in the language.
- Many bugs, most of which appeared in the new features, have been removed.
- The quality of the generated code has been improved.

`Release 2.0 (November 2001).`

There are *many* changes compared to the previous release (CLEAN 1.3.x). We have added many new features in CLEAN 2.0 we hope you will like.

- CLEAN 2.0 has multi-parameter type constructor classes. See Section 6.4.
- CLEAN 2.0 has universally quantified data types and functions (rank 2). See Section 3.7.4 and 5.1.4.
- The explicit import mechanism has been refined. One can now more precisely address what to import and what not. See 2.5.1.
- Cyclic depedencies between definition modules are allowed. This makes it easier to define implementation modules that share definitions. See 2.5.1.
- Definitions in a definition module need not to be repeated in the corresponding implementation module anymore. See 2.4.
- Due to multi-parameter type constructor classes a better incorporation of the type Array could be made. See 4.4.
- CLEAN 2.0 offers an hybrid type system: one can have statically and dynamically typed objects (**Dynamics**). A statically typed expression can be changed into a dynamically typed one and backwards. The type of a `Dynamic` can be inspected via a pattern match, one can ensure that `Dynamics` fit together by using run-time type unification, one can store a `Dynamic` into a file with one function call or read a `Dynamic` stored by another CLEAN application. `Dynamics` can be used to store and retrieve information without the need for writing parsers, it can be used to exchange data and code (!) between applications in a type safe manner. `Dynamics`make it easy to create mobile code, create plug-ins or create a persistent store. The CLEAN run-time system has been extended to support dynamic type checking, dynamic type unification, lazy dynamic linking and just-in-time code generation (See Chapter 8).

- There is special syntax and support for strict and unboxed lists. One can easily change from lazy to strict and backwards. Overloaded functions can be defined which work for any list (lazy, strict or unboxed). See 4.2. One can write functions like `==`, `map`, `foldr` in a generic way. The `generic functions` one can define can work on higher order kinds. With kind indexed functions one can indicated which kind is actually meant (see Chapter 7). A generic definition can be specialized for a certain concrete type.
- The CLEAN system has been changed and extended: a new version of the CLEAN IDE, a new version of the run-time-system, and a dynamic linker is included. See 8.3.
- CLEAN 2.0 comes with an integrated proof system (Sparkle), all written in CLEAN of course. See http://www.cs.kun.nl/Sparkle.
- CLEAN 2.0 is open source. All source code will be made available on the net.

We have also removed things:

- We do not longer support annotations for concurrent evaluations ({P} and {I} annotations. However, we are working on a library that will support distributed evaluation of CLEAN expressions using Dynamics (see Van Weelden and Plasmeijer, 2002).
- There is no strict let-before expression (`let!`) anymore in CLEAN 2.x. You still can enforce strict evaluation using the strict hash let (`#!`).
- One cannot specify default instances anymore that could be used to disambiguate possible ambiguous internal overloading. Disambiguating can be done by explicitely specifying the required type.

There is also some bad news:

- Due to all these changes CLEAN 2.0 is *not* upwards compatible with CLEAN 1.3.x. Many things are the same but there are small differences as well. So, one has to put some effort in porting a CLEAN 1.3.x application to CLEAN 2.0. The most important syntactical differences are described below. Note that we do no longer support CLEAN 1.3.
- The CLEAN 1.3 compiler is written in C. The CLEAN 2.0 compiler has been rewritten from scratch in CLEAN. The internal structure of the new compiler is a better than the old one, but the new compiler has become a bit slower than the previous C version as well. Large programs will take about 1.7 times as much time to compile (which is still pretty impressive for a lazy functional language).

Syntactic differences between Clean 1.3 and Clean 2.0

CLEAN 2.x is not downward compatible with CLEAN 1.3.x. Probably you have to change your 1.3.x sources to get them through the CLEAN 2.x compiler.

Differences in Expression Syntax

There is no *strict let* expression (`let!`) anymore in CLEAN 2.x. You still can enforce strict evaluation using the strict hash let (`#!`).

Differences in the Type System

- For *multiparameter type* classes a small change in the syntax for instance definitions was necessary. In CLEAN 1.3.x it was assumed that every instance definition only has one type argument. So in the following 1.3.x instance definition

```
instance c T1 T2
```

the type `(T1 T2)` was meant (the type `T1` with the argument `T2`). This should be written in CLEAN 2.x as

```
instance c (T1 T2)
```

otherwise `T1` and `T2` will be interpreted as two types.

- The *type* `Array` has changed. In CLEAN 2.x the `Array` class has become a multiparameter class, whose first argument type is the array and whose second argument type is the array element (see ??). Therefore a 1.3 definition like

```
MkArray:: !Int (Int -> e) ->.(a e) | Array a & ArrayElem e
MkArray i f = {f j \ j <- [0..i-1]}
```

  becomes in CLEAN 2.x:

```
MkArray:: !Int (Int -> e) ->.(a e) | Array a e
MkArray i f = {f j \ j <- [0..i-1]}
```

- There is a difference in *resolving overloading*. Consider the following code:

```
class c a :: a -> a

instance c [Int]
   where
     c [1] = [2]

f [x:xs]
  = c xs
```

  Although this is accepted by CLEAN 1.3.x, CLEAN 2.x will complain: "Overloading error [...,..,f]: c no instance available of type [a]." The CLEAN 2.x compiler applies no type unification after resolving overloading. So `c` is in function `f` applied to a list with a polymorph element type (`[a]`). And this is considered to be different from the instance type `[Int]`. If you give f the type `[Int] -> [Int]` the upper code will be accepted.

- CLEAN 2.x handles *uniqueness attributes in type synonyms* different than CLEAN 1.3.x. Consider the following definitions:

```
:: ListList a :== [[a]]

f :: *(ListList *{Int}) -> *{Int}
f [[a]] = { a & [0]=0 }
```

  In CLEAN 1.3.x the `ListList` type synonym was expanded to

```
f :: *[*[*{Int}]] -> *{Int}
```

  which is correct in CLEAN 1.3.x. However, CLEAN 2.x expands it to

```
f :: *[[*{Int}]] -> *{Int}
```

  This yields a uniqueness error in CLEAN 2.x because the inner list is shared but contains a unique object. This problem happens only with type synonyms that have attributes "inbetween". An "inbetween" attribute is neither the "root" attribute nor the attribute of an actual argument. E.g. with the upper type synonym, the formal argument "`a`" is substituted with `*{Int}`. Note that also the "`*`" is substituted for "`a`". Because we wrote `*(ListList ...)` the root attribute is "`*`". The result of expanding `*(ListList *{Int})` is `*[u:[*{Int}]]`. "`u`" is an attribute "inbetween" because it is neither the root attribute nor the attribute of a formal argument. Such attributes are made _non_unique_ in CLEAN 2.x and this is why the upper code is not accepted. The code will be accepted if you redefine `ListList` to

```
:: ListList a :== [*[a]]
```

- *Anonymous uniqueness attributes in type contexts* are not allowed in CLEAN 2.x. So in the following function type simply remove the point.

```
f :: a | myClass .a
```

- The `String` type has become a *predefined type*. As a consequence you cannot import this type explicitly anymore. So:

```
from StdString import :: String
```

  is not valid.

- There was a bug in the uniqueness typing system of CLEAN 1.3: Records or data constructors could have existentially quantified variables, whose uniqueness attribute did _not_ propagate. This bug has been solved in CLEAN 2.x. As a consequence, the 2.x compiler might complain about your program where the 1.3.x compiler was happy. The problem might occur when you use the object I/O library and you use objects with a uniquely attributed local state. Now the object becomes unique as well and may not be shared anymore.

Differences in the Module System

- The syntax and semantics of *explicit import statements* has been completely revised. With CLEAN 2.x it is possible to discriminate the different namespaces in import statements. In CLEAN 1.3.x the following statement

```
from m import F
```

  could have caused the import of a *function* `F` together with a *type* `F` and a *class* `F` with all its instances from m. In CLEAN 2.x one can precisely describe from which name space one wants to import (see 2.5.2). For example, the following import statement

```
from m import  F,
               :: T1, :: T2(..), :: T3(C1, C2), :: T4{..}, :: T5{field1, field2},
               class C1, class C2(..), class C3(mem1, mem2)
```

  causes the following declarations to be imported: the *function* or *macro* `F`, the *type* `T1`, the *algebraic type* `T2` with *all* it's constructors that are exported by `m`, the *algebraic type* `T3` with it's constructors `C1` and `C2`, the *record type* `T4` with *all* it's fields that are exported by `m`, the *record type* `T5` with it's fields `field1` and `field2`, the *class* `C1`, the *class* `C2` with all it's members that are exported by `m`, the *class* `C3` with it's members `mem1` and `mem2`.

Previous Releases. The first release of CLEAN was publicly available in 1987 and had version number 0.5 (we thought half of the work was done, ;-)). At that time, CLEAN was only thought as an intermediate language. Many releases followed. One of them was version 0.8 which is used in the Plasmeijer & Van Eekelen Bible (Adisson-Wesley, 1993). Version 1.0 was the first mature version of CLEAN.

Chapter 1

Basic Semantics

The semantics of CLEAN is based on *Term Graph Rewriting Systems* (Barendregt, 1987; Plasmeijer and Van Eekelen, 1993). This means that functions in a CLEAN program semantically work on *graphs* instead of the usual *terms*. This enabled us to incorporate CLEAN's typical features (definition of cyclic data structures, lazy copying, uniqueness typing) which would otherwise be very difficult to give a proper semantics for. However, in many cases the programmer does not need to be aware of the fact that he/she is manipulating graphs. Evaluation of a CLEAN program takes place in the same way as in other lazy functional languages. One of the "differences" between CLEAN and other functional languages is that when a variable occurs more than once in a function body, the semantics *prescribe* that the actual argument is shared (the semantics of most other languages do not prescribe this although it is common practice in any implementation of a functional language). Furthermore, one can label any expression to make the definition of cyclic structures possible. So, people familiar with other functional languages will have no problems writing CLEAN programs.

When larger applications are being written, or, when CLEAN is interfaced with the non-functional world, or, when efficiency counts, or, when one simply wants to have a good understanding of the language it is good to have some knowledge of the basic semantics of CLEAN which is based on term graph rewriting. In this chapter a short introduction into the basic semantics of CLEAN is given. An extensive treatment of the underlying semantics and the implementation techniques of CLEAN can be found in Plasmeijer and Van Eekelen (1993).

1.1 Graph Rewriting

A CLEAN *program* basically consists of a number of *graph rewrite rules* (*function definitions*) which specify how a given *graph* (the *initial expression*) has to be *rewritten*.

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

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

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

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

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

A pattern *partially matches* a graph if firstly the symbol of the root of the pattern equals the symbol of the root of the graph and secondly in positions where symbols in the pattern are not syntactically equal to symbols in the graph, the corresponding sub-graph is a redex or the sub-graph itself is partially matching a rule. A graph is in *strong root normal form* if the graph does not partially match any rule. It is decidable whether or not a graph is in strong root normal form. A graph in strong root normal form does not partially match any rule, so it is also in root normal form.

The default reduction strategy used in CLEAN is the *functional reduction strategy*. Reducing graphs according to this strategy resembles very much the way execution proceeds in other lazy functional languages: in the standard lambda calculus semantics the functional strategy corresponds to normal order reduction. On graph rewrite rules the functional strategy proceeds as follows: if there are several rewrite rules for a particular function, the rules are tried in textual order; patterns are tested from left to right; evaluation to strong root normal form of arguments is forced when an actual argument is matched against a corresponding non-variable part of the pattern. A formal definition of this strategy can be found in (Toyama *et al.*, 1991).

1.1.1 A Small Example

Consider the following CLEAN program:

```
Add Zero z          =    z                        (1)
Add (Succ a) z      =    Succ (Add a z)           (2)

Start               =    Add (Succ o) o
                         where
                             o = Zero             (3)
```

In CLEAN a distinction is between function definitions (graph rewriting rules) and graphs (constant definitions). A semantic equivalent definition of the program above is given below where this distinction is made explicit ("=>" indicates a rewrite rule whereas "=:" is used for a constant (*sub-*) *graph* definition

```
Add Zero z          =    z                        (1)
Add (Succ a) z      =    Succ (Add a z)           (2)

Start               =    Add (Succ o) o
                         where
                             o =: Zero            (3)
```

These rules are internally translated to a semantically equivalent set of rules in which the graph structure on both left-hand side as right-hand side of the rewrite rules has been made explicit by adding node-id's. Using the set of rules with explicit node-id's it will be easier to understand what the meaning is of the rules in the graph rewriting world.

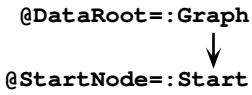
```
x =: Add y z
y =: Zero         =>    z                        (1)
x =: Add y z
y =: Succ a       =>    m =: Succ n
                        n =: Add a z             (2)

x =: Start        =>    m =: Add n o
                        n =: Succ o
                        o =: Zero                (3)
```

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

The initial graph in linear notation:

```
@DataRoot     =: Graph @StartNode
@StartNode    =: Start
```

The initial graph in pictorial notation:



To distinguish the node-id's appearing in the rewrite rules from the node-id's appearing in the graph the latter always begin with a "`@`".

The initial graph is rewritten until it is in normal form. Therefore a CLEAN program must at least contain a "*start rule*" that matches this initial graph via a pattern. The right-hand side of the start rule specifies the actual computation. In this start rule in the left-hand side the symbol `Start` is used. However, the symbols `Graph` and `Initial` (see 1.2) are internal, so they cannot actually be addressed in any rule.

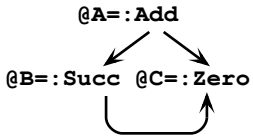
The patterns in rewrite rules contain *formal node-id's*. During the matching these formal nodeid's are mapped to the *actual node-id's*of the graph After that the following semantic actions are performed:

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

The contractum in linear notation:

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

The contractum in pictorial notation:

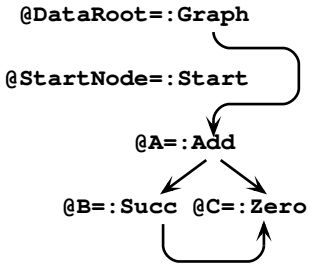


All applied occurrences of `@StartNode` will be replaced by occurrences of `@A`. The graph after rewriting is then:

The graph after rewriting:

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

Pictorial notation:

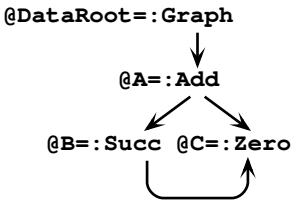


This completes one rewrite. All nodes that are not accessible from `@DataRoot` are garbage and not considered any more in the next rewrite steps. In an implementation once in a while garbage collection is performed in order to reclaim the memory space occupied by these garbage nodes. In this example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

The graph after garbage collection:

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

Pictorial notation:

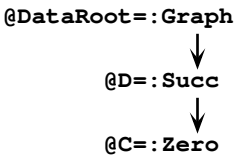


The graph accessible from `@DataRoot` still contains a redex. It matches rule 2 yielding the expected normal form:

The final graph:

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

Pictorial notation:



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

1.2 Global Graphs

Due to the presence of global graphs in CLEAN the initial graph in a specific CLEAN program is slightly different from the basic semantics. In a specific CLEAN program the initial graph is defined as:

```
@DataRoot    =: Graph @StartNode @GlobId1 @GlobId2 ... @GlobIdn
@StartNode   =: Start
@GlobId1     =: Initial
@GlobId2     =: Initial
...
@GlobIdn     =: Initial
```

The root of the initial graph will not only contain the node-id of the start node, the root of the graph to be rewritten, but it will also contain for each *global graph* (see 10.2) a reference to an initial node (initialized with the symbol `Initial`). All references to a specific global graph will be references to its initial node or, when it is rewritten, they will be references to its reduct.

Chapter 2

Modules and Scopes

A CLEAN program is composed out of modules. Each module is stored in a file that contains CLEAN source code. There are implementation modules and definition modules, in the spirit of Modula-2 (**Wirth**, 1982). This module system is used for several reasons.

- First of all, the module structure is used to *control the scope of definitions*. The basic idea is that definitions only have a meaning in the implementation module they are defined in unless they are exported by the corresponding definition module.
- Having the exported definitions collected in a separate definition module has as advantage that one in addition obtains a *self-contained interface document* one can reach out to others. The definition module is a document that defines which functions and data types can be used by others without revealing uninteresting implementation details.
- Furthermore, the module structure enables *separate compilation* that heavily reduces compilation time. If the definition module stays the same, a change in an implementation module only will cause the recompilation of that implementation module. When the definition module is changed as well, only those implementation modules that are affected by this change need to be recompiled.

In this Chapter we explain the module structure of CLEAN and the influence it has on the scope of definitions. New scopes can also be introduced inside modules. This is further explained in the Chapters 2 and 3.

In the pictures in the subsections below nested boxes indicate nested scopes.

2.1 Identifiers, Scopes and Name Spaces

2.1.1 Naming Conventions of Identifiers

In CLEAN we distinguish the following kind of *identifiers*.

```
ModuleName              =  LowerCaseId  |  UpperCaseId  |  ModuleDirectoryName . ModuleName
ModuleDirectoryName     =  LowerCaseId  |  UpperCaseId
FunctionName            =  LowerCaseId  |  UpperCaseId  |  SymbolId
ConstructorName         =               |  UpperCaseId  |  SymbolId
SelectorVariable        =  LowerCaseId
Variable                =  LowerCaseId
MacroName               =  LowerCaseId  |  UpperCaseId  |  SymbolId
FieldName               =  LowerCaseId
TypeName                =               |  UpperCaseId  |  SymbolId
TypeVariable            =  LowerCaseId
UniqueTypeVariable      =  LowerCaseId
ClassName               =  LowerCaseId  |  UpperCaseId  |  SymbolId
MemberName              =  LowerCaseId  |  UpperCaseId  |  SymbolId
```

```
LowerCaseId             =  LowerCaseChar{IdChar}
UpperCaseId             =  UpperCaseChar{IdChar}
SymbolId                =  {SymbolChar}+
```

```
LowerCaseChar           =  a | b | c | d | e | f | g | h | i | j | k | l | m
                        |  n | o | p | q | r | s | t | u | v | w | x | y | z
UpperCaseChar           =  A | B | C | D | E | F | G | H | I | J | K | L | M
                        |  N | O | P | Q | R | S | T | U | V | W | X | Y | Z
SymbolChar              =  ~ | @ | # | $ | % | ^ | ? | ! | :
                        |  + | - | * | < | > | \ | / | | | & | =
IdChar                  =  LowerCaseChar
                        |  UpperCaseChar
                        |  Digit
                        |  _ | `
```

```
Digit                   =  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The convention used is that variables always start with a lowercase character while constructors and types always start with an uppercase character. The other identifiers can either start with an uppercase or a lowercase character. Notice that for the identifiers names can be used consisting of a combination of lower and/or uppercase characters but one can also define identifiers constructed from special characters like `+`, `<`, etc. (see Appendix A). These two kinds of identifiers cannot be mixed. This makes it possible to leave out white space in expressions like `a+1` (same as `a + 1`).

Some names may be prepended with the module name from which they are imported using a qualified import. These are qualified identifiers.

```
QFunctionName      =  QLowerCaseId  |  QUpperCaseId  |  QSymbolId
QConstructorName   =                  QUpperCaseId  |  QSymbolId
QTypeName          =                  QUpperCaseId  |  QSymbolId
QClassName         =  QLowerCaseId  |  QUpperCaseId  |  QSymbolId
```

```
QLowerCaseId       =  [ModuleQualifier]LowerCaseId
QUpperCaseId       =  [ModuleQualifier]UpperCaseId
QSymbolId          =  [ModuleQualifier space]SymbolId
ModuleQualifier    =  'ModuleName'.
```

2.1.2 Scopes and Name Spaces

The *scope* is the program region in which definitions (e.g. function definition, class definition, macro definition, type definition) with the identifiers introduced (e.g. function name, class name, class variable, macro name, type constructor name, type variable name) have a meaning.

It must be clear from the context to which definition an identifier is referring. If all identifiers in a scope have different names than it will always be clear which definition is meant. However, one generally wants to have a free choice in naming identifiers. If identifiers belong to different *name spaces* no conflict can arise even if the same name is used. In CLEAN the following name spaces exist:

- ModuleNames form a name space;
- FunctionNames, ConstructorNames, SelectorVariables, Variables and MacroNames form a name space;
- FieldNames form a name space;
- TypeNames, TypeVariables and UniqueTypeVariables form a name space;
- ClassNames form a name space.

So, it is allowed to use the same identifier name for different purposes as long as the identifier belongs to different name spaces.

Identifiers belonging to the same name space must all have different names within the same scope. Under certain conditions it is allowed to use the same name for different functions and operators (overloading, see Chapter 6).

2.1.3 Nesting of Scopes

Reusing identifier names is possible by introducing a new scope level. Scopes can be nested: within a scope a new *nested scope* can be defined. Within such a nested scope new definitions can be given, new names can be introduced. As usual it is allowed in a nested scope to redefine definitions or re-use names given in a surrounding scope: When a name is re-used the old name and definition is no longer in scope and cannot be used in the new scope. A definition given or a name introduced in a (nested) scope has no meaning in surrounding scopes. It has a meaning for all scopes nested within it (unless they are redefined within such a nested scope).

2.2 Modular Structure of Clean Programs

A CLEAN program consists of a collection of *definition modules* and *implementation modules*. An implementation module and a definition module *correspond* to each other if the names of the two modules are the same. The basic idea is that the definitions given in an implementation module only have a meaning in the module in which they are defined unless these definitions are exported by putting them into the corresponding definition module. In that case the definitions also have a meaning in those other modules in which the definitions are imported (see 2.5).

```
CleanProgram          =  {Module}+
Module                =  DefinitionModule
                      |  ImplementationModule
DefinitionModule      =  definition module ModuleName ;
                         {DefDefinition}
                      |  system module ModuleName ;
                         {DefDefinition}
ImplementationModule  =  [implementation] module ModuleName ;
                         {ImplDefinition}
```

- An executable CLEAN program consists at least of one implementation module, the *main* or *start module*, which is the top-most module (*root module*) of a CLEAN program.
- Each CLEAN module has to be put in a separate file.
- The name of a module (i.e. the module name) should be the same as the name of the file (minus the suffix) in which the module is stored.
- A *definition* module should have *.dcl* as suffix; an *implementation* module should have *.icl* as suffix.
- A definition module can have at most one corresponding implementation module.
- Every implementation module (except the main module, see 2.3.1) must have a corresponding definition module.

2.3 Implementation Modules

2.3.1 The Main or Start Module

- In the main module a `Start` rule has to be defined (see Chapter 1).
- Only in the main module one can leave out the keyword implementation in the module header. In that case the `implementation module` does not need to have a corresponding definition module (which makes sense for a topmost module).

A very tiny but complete CLEAN program consisting of one implementation module.

```
module hello

Start = "Hello World!"
```

Evaluation of a CLEAN *program* consists of the evaluation of the application defined in the right-hand side of the `Start` *rule* to normal form (see Chapter 1). The right-hand side of the `Start` rule is regarded to be the *initial expression* to be computed.

It is allowed to have a `Start` rule in other implementation modules as well. This can be handy for testing functions defined in such a module: to evaluate such a `Start` rule simply generate an application with the module as root and execute it. In the CLEAN IDE one can specify which module has to be regarded as being the root module.

The definition of the left-hand side of the Start rule consists of the *symbol* `Start` with one optional argument (of type `*World`), which is the environment parameter, which is necessary to write interactive applications.

A CLEAN program can run in two modes.

I/O Using the Console

The first mode is a *console mode*. It is chosen when the Start rule is defined as a *nullary* function.

```
Start:: TypeOfStartFunction
Start = ...                          // initial expression
```

In the console mode, that part of the *initial expression* (indicated by the right-hand side of the Start rule), which is in *root normal form* (also called the head normal form or root stable form), is printed as soon as possible. The console mode can be used for instance to test or debug functions.

In the CLEAN IDE one can choose to print the result of a `Start` expression *with* or *without* the data constructors.

For example, the initial expression

```
Start:: String
Start = "Hello World!"
```

in mode "show data constructors" will print: `"Hello World!"`, in mode "don't show data constructors" it will print: `Hello World!`

The second mode is the *world mode*. It is chosen when the optional additional parameter (which is of type `*World`) is added to the `Start` rule and delivered as result.

```
Start:: *World -> *World
Start = ...                          // initial expression returning a changed world
```

The world which is given to the initial expression is an *abstract data structure*, an *abstract world* of type *World which models *the concrete physical world* as seen from the program. The abstract world can in principle contain *anything* what a functional program needs to interact during execution with the concrete world. The world can be seen as a *state* and modifications of the world can be realized via *state transition functions* defined on the world or a part of the world. By requiring that these state transition functions work on a *unique* world the modifications of the abstract world can directly be realized in the real physical world, without loss of efficiency and without losing referential transparency (see Chapter 9)

The concrete way in which the world can be handled in CLEAN is determined by the system programmer. One way to handle the world is by using the predefined CLEAN I/O library, which can be regarded as a platform independent mini operating system. It makes it possible to do file I/O, window based I/O, dynamic process creation and process communication in a pure functional language in an efficient way. The definition of the I/O library is treated in a separate document (Object IO tutorial, Achten *et al.*, 1997).

2.3.2 Scope of Global Definitions in Implementation Modules

In an implementation module the following global definitions can be specified in *any* order.

```
ImplDefinition  =  ImportDef          // see 2.5
                |  FunctionDef        // see Chapter 3
                |  GraphDef           // see 3.6
                |  MacroDef           // see 10.3
                |  TypeDef            // see Chapter 5
                |  ClassDef           // see Chapter 6
                |  GenericsDef        // see Chapter 7
```

Definitions on the *global* level (= outermost level in the module,) have in principle the whole implementation module as scope (see Figure 2.1).

Figure 2.1 (Scope of global definitions inside an implementation module).

```
implementation module XXX

:: TypeName typevars = type_expression          // definition of a new type

functionName:: type_of_args -> type_of_result   // definition of the type of a function
functionName args = expression                  // definition of a function

selector = expression                           // definition of a constant graph

class className = expression                    // definition of a class

macroName args :==  expression                  // definition of a macro
```

Types can only be defined globally (see Chapter 5) and therefore always have a meaning in the whole implementation module. Type variables introduced on the left-hand side of a (algebraic, record, synonym, overload, class, instance, function, graph) type definition have the right-hand side of the type definition as scope.

Functions, the type of these functions, constants (selectors) and macro's can be defined on the *global* level as well as on a *local* level in nested scopes. When defined globally they have a meaning in the whole implementation module. Arguments introduced on the left-hand side of a definition (formal arguments) only have a meaning in the corresponding right-hand side.

Functions, the type of these functions, constants (selectors) and macro's can also be defined locally in a new scope. However, new scopes can only be introduced at certain points. In functional languages local definitions are by tradition defined by using *let*-expressions (definitions given *before* they are used in a certain expression, nice for a bottom-up style of programming) and *where*-blocks (definitions given *afterwards*, nice for a top-down style of programming). These constructs are explained in detail in Chapter 3.

2.3.3 Begin and End of a Definition: the Layout Rule

CLEAN programs can be written in two modes: layout sensitive mode 'on' and 'off'. The layout sensitive mode is switched off when a semi-colon is specified after the module name. In that case each definition has to be ended with a semicolon '`;`'. A new scope has to begin with '`{`' and ends with a '`}`'. This mode is handy if CLEAN code is generated automatically (e.g. by a compiler).

Example of a CLEAN program *not* using the layout rule.

```
module primes;

import StdEnv;

primes:: [Int];
primes = sieve [2..];
where
{   sieve:: [Int] -> [Int]; sieve [pr:r] = [pr:sieve (filter pr r)];

    filter:: Int [Int] -> [Int];
    filter pr [n:r] | n mod pr == 0 = filter pr r;
    | otherwise     = [n:filter pr r];
}
```

Programs look a little bit old fashion C-like in this way. Functional programmers generally prefer a more mathematical style. Hence, as is common in modern functional languages, there is a layout rule in CLEAN. When a semicolon does not end the header of a module, a CLEAN program has become layout sensitive. The *layout rule* assumes the omission of the semi-colon ('`;`') that ends a definition and of the braces ('`{`' and '`}`') that are used to group a list of definitions. These symbols are automatically added according to the following rules:

In *layout sensitive mode* the indentation of the first lexeme after the keywords **`let`**, `#`, `#!`, **`of`**, **`where`**, or **`with`** determines the indentation that the group of definitions following the keyword has to obey. Depending on the indentation of the first lexeme on a subsequent line the following happens. A new definition is assumed if the lexeme starts on the same indentation (and a semicolon is inserted). A previous definition is assumed to be continued if the lexeme is indented more. The group of definitions ends (and a close brace is inserted) if the lexeme is indented less. Global definitions are assumed to start in column 0.

We strongly advise to write programs in layout sensitive mode. *For reasons of portability it is assumed that a tab space is set to 4 white spaces and that a non-proportional font is used.*

Same program using the layout sensitive mode.

```
module primes

import StdEnv

primes:: [Int]
primes = sieve [2..]
where
    sieve:: [Int] -> [Int]
    sieve [pr:r] = [pr:sieve (filter pr r)]

    filter:: Int [Int] -> [Int]
    filter pr [n:r] | n mod pr == 0 = filter pr r
    | otherwise     = [n:filter pr r]
```

2.4 Definition Modules

The definitions given in an implementation module only have a meaning in the module in which they are defined. If you want to *export* a definition, you have to specify the definition in the corresponding definition module. Some definitions can only appear in implementation modules, not in definition modules. The idea is to hide the actual implementation from the outside world. The is good for software engineering reasons while another advantage is that an implementation module can be recompiled separately without a need to recompile other modules. Recompilation of other modules is only necessary when a definition module is changed. All modules depending on the changed module will have to be recompiled as well. Implementations of functions, graphs and class instances are therefore only allowed in *implementation* modules. They are exported by only specifying their type definition in the definition module. Also the right-hand side of any type definition can remain hidden. In this way an abstract data type is created (see 5.4).

In a definition module the following global definitions can be given in *any* order.

DefDefinition	=	ImportDef	// see 2.5
	|	FunctionExportTypeDef	// see 3.7
	|	MacroDef	// see 10.3
	|	TypeDef	// see Chapter 5
	|	ClassExportDef	// see Chapter 6
	|	TypeClassInstanceExportDef	// see 6.10
	|	GenericExportDef	// see Chapter 7

- The definitions given in an implementation module only have a meaning in the module in which they are defined (see 2.3) unless these definitions are exported by putting them into the corresponding definition module. In that case they also have a meaning in those other modules in which the definitions are imported (see 2.5).
- In the corresponding implementation module all exported definitions have to get an appropriate implementation (this holds for functions, abstract data types, class instances).
- An *abstract data type* is exported by specifying the left-hand side of a type rule in the definition module. In the corresponding implementation module the abstract type *has to be defined again* but then right-hand side has to be defined as well. For such an abstract data type only the name of the type is exported but not its definition.
- A *function*, global *graph* or *class instance* is exported by defining the type header in the definition module. For optimal efficiency it is recommended also to specify strictness annotations (see 10.1). For library functions it is recommended also to specify the uniqueness type attributes (see Chapter 9). The implementation of a function, a graph, a class instance has to be given in the corresponding implementation module.
- Although it is not required anymore to repeat an exported definition in the corresponding implementation module, it is a good habit to do so to keep the implementation module readable. If a definition is repeated, the definition given in the definition module and in the implementation module should be the same (modulo variable names).

Definition module.

```
definition module ListOperations

::complex                          // abstract type definition

re:: complex -> Real               // function taking the real part of complex number
im:: complex -> Real               // function taking the imaginary part of complex

mkcomplex:: Real Real -> Complex   // function creating a complex number
```

corresponding implementation module:

```
implementation module ListOperations

::complex :== (!Real,!Real)        // a type synonym

re:: complex -> Real               // type of function followed by its implementation
re (frst,_) = frst

im:: complex -> Real
im (_,scnd) = scnd

mkcomplex:: Real Real -> Complex
mkcomplex frst scnd = (frst,scnd)
```

2.5 Importing Definitions

Via an *import statement* a definition *exported* by a definition module (see 2.4) can be *imported* into any other (definition or implementation) module. There are two kinds of import statements, *explicit* imports and *implicit* imports.

ImportDef	=	ImplicitImportDef
	|	ExplicitImportDef

A module *depends* on another module if it imports something from that other module. Cyclic dependencies are allowed.

2.5.1 Explicit Imports of Definitions

Explicit imports are import statements in which the modules to import from as well as the identifiers indicating the definitions to import are explicitly specified. All identifiers explicitly being imported in a definition or implementation module will be included in the global scope level (= outermost scope, see 2.3.2) of the module that does the import.

```
ExplicitImportDef  =  from ModuleName import [qualified] {Imports}-list ;
Imports            =  FunctionName
                   |  ::TypeName [ConstructorsOrFields]
                   |  class ClassName [Members]
                   |  instance ClassName {SimpleType}+
                   |  generic FunctionName
ConstructorsOrFields = (..)
                   |  ({ConstructorName}-list)
                   |  {..}
                   |  {{FieldName}-list}
Members            =  (..)
                   |  ({MemberName}-list)
```

One can import functions or macro's, types with optionally their corresponding constructors, record types with optionally their corresponding fieldnames, classes, instances of classes and generic functions. The syntax makes it possible to discriminate between the different namespaces that exist in CLEAN (see 2.1.2)

Example of an explicit import.

```
implementation module XXX

from m import     F,
                  :: T1, :: T2(..), :: T3(C1, C2), :: T4{..}, :: T5{field1, field2},
                  class C1, class C2(..), class C3(mem1, mem2),
                  instance C4 Int, generic g
```

With the import statement the following definitions exported by module `m` are imported in module `XXX`: the *function* or *macro* `F`, the *type* `T1`, the *algebraic* type `T2` with *all* it's constructors that are exported by m, the *algebraic type* `T3` with it's constructors `C1` and `C2`, the *record type* `T4` with *all* it's fields that are exported by m, the *record type* `T5` with it's fields `field1` and `field2`, the *class* `C1`, the *class* `C2` with all it's members that are exported by `m`, the *class* `C3` with it's members `mem1` and `mem2`, the instance of *class* `C4` defined on integers, the generic function `g`.

Importing identifiers can cause error messages because the imported identifiers may be in conflict with other identifiers in this scope (remember that identifiers belonging to the same name space must all have different names within the same scope, see 2.1). This problem can be solved by renaming the internally defined identifiers or by renaming the imported identifiers (eg by adding an additional module layer just to rename things).

2.5.2 Implicit Imports of Definitions

```
ImplicitImportDef  =  import [qualified] {ModuleName}-list ;
```

Implicit imports are import statements in which only the module name to import from is mentioned. In this case *all* definitions that are *exported* from that module are imported as well as *all* definitions that on their turn are *imported* in the indicated definition module, and so on. So, all related definitions from various modules can be imported with one single import.
This opens the possibility for definition modules to serve as a kind of *'pass-through'* module. Hence, it is meaningful to have definition modules with import statements but without any definitions.

Example of an implicit import: all (arithmetic) rules which are predefined can be imported easily with one import statement.

```
import MyStdEnv
```

importing implicitly all definitions imported by the definition module '`MyStdEnv`' which is defined below:

```
definition module MyStdEnv

import StdBool, StdChar, StdInt, StdReal, StdString
```

All identifiers implicitly being imported in a definition or implementation module will be included in the global scope level (= outermost scope, see 2.3.2) of the module that does the import.

Importing identifiers can cause error messages because the imported identifiers may be in conflict with other identifiers in this scope (remember that identifiers belonging to the same name space must all have different names within the same scope, see 2.1). This problem can be solved by renaming the internally defined identifiers or by renaming the imported identifiers (eg by adding an additional module layer just to rename identifiers).

2.6 System Definition and Implementation Modules

System modules are special modules. A *system definition module* indicates that the corresponding implementation module is a *system implementation module* which does not contain ordinary CLEAN rules. In system implementation modules it is allowed to define *foreign functions*: the bodies of these foreign functions are written in another language than CLEAN. System implementation modules make it possible to create interfaces to operating systems, to file systems or to increase execution speed of heavily used functions or complex data structures. Typically, predefined function and operators for arithmetic and File I/O are implemented as system modules.

System implementation modules may use machine code, C-code, abstract machine code (PABC-code) or code written in any other language. What exactly is allowed depends on the CLEAN compiler used and the platform for which code is generated. The keyword **code** is reserved to make it possible to write CLEAN programs in a foreign language. This is not treated in this reference manual.

When one writes system implementation modules one has to be very careful because the correctness of the functions can no longer be checked by the CLEAN compiler. Therefore, the programmer is now responsible for the following:

- The function must be correctly typed.
- When a function destructively updates one of its (sub-)arguments, the corresponding type of the arguments should have the uniqueness type attribute. Furthermore, those arguments must be strict.

Chapter 3

Defining Functions and Constants

In this Chapter we explain how functions (actually: *graph rewrite rules*) and constants (actually: graph expressions) are defined in CLEAN. The body of a function consists of an (root) expression (see 3.4). With help of patterns (see 3.2) and guards (see 3.3) a distinction can be made between several alternative definitions for a function. Functions and constants can be defined locally in a function definition. For programming convenience (forcing evaluation, observation of unique objects and threading of sequencial operations) a special let construction is provided (see 3.5.1). The typing of functions is discussed in Section 3.7. For overloaded functions see Chapter 6. For functions working on unique datatypes see Chapter 9.

3.1 Functions

```
FunctionDef          =  [FunctionTypeDef]                  // see Chapter 4 for typing functions
                        DefOfFunction
DefOfFunction        =  {FunctionAltDef ;}+
FunctionAltDef       =  Function {Pattern}                 // see 3.2 for patterns
                        {GuardAlt}
                        {LetBeforeExpression}              // see 3.5.4
                        FunctionResult
                        [LocalFunctionAltDefs]             // see 3.5
FunctionResult       =  =[>] FunctionBody
                     |  | Guard GuardRhs                   // see 3.3 for guards
GuardAlt             =  {LetBeforeExpression} | BooleanExpr GuardRhs
GuardRhs             =  {GuardAlt} {LetBeforeExpression} = [>] FunctionBody
                     |  {GuardAlt} {LetBeforeExpression} | otherwise GuardRhs
Function             =  FunctionName                       // ordinary function
                     |  (FunctionName)                     // operator function used prefix
FunctionBody         =  RootExpression ;                   // see 3.4
                        [LocalFunctionDefs]                // see 3.5
```

A *function definition* consists of one or more definition of a *function alternative* (rewrite rule). On the left-hand side of such a function alternative a *pattern* can be specified which can serve a whole sequence of *guarded function bodies* (called the *rule alternatives*) The root expression (see 3.4) of a particular rule alternative is chosen for evaluation when

- the patterns specified in the formal arguments are matching the corresponding actual arguments of the function application (see 3.2) *and*
- the optional *guard* (see 3.3) specified on the right-hand side evaluates to `True`.

The alternatives are tried in textual order. A function can be preceded by a definition of its type (Section 3.7).

- Function definitions are only allowed in implementation modules (see 2.3).
- It is required that the function alternatives of a function are textually grouped together (separated by semi-colons when the layout sensitive mode is not chosen).
- Each alternative of a function must start with the same function symbol.
- A function has a fixed arity, so in each rule the same number of formal arguments must be specified. Functions can be used curried and applied to any number of arguments though, as usual in higher order functional languages.
- The function name must in principle be different from other names in the same name space and same scope (see 2.1). However, it is possible to overload functions and operators (see Chapter 6).

Example of function definitions in a CLEAN module.

```
module example                                  // module header

import StdInt                                   // implicit import

map:: (a -> b) [a] -> [b]                       // type of map
map f list = [f e \\ e <- list]                 // definition of the function map

Start:: [Int]                                   // type of Start rule
Start = map square [1..1000]                    // definition of the Start rule
```

An *operator* is a *function with arity two* that can be used as infix operator (brackets are left out) or as ordinary prefix function (the operator name preceding its arguments has to be surrounded by brackets). The *precedence* (`0` through `9`) and *fixity* (**`infixl`**eft, **`infixr`**ight, **`infix`**) that can be defined in the type definition (see 3.7.1) of the operators determine the priority of the operator application in an expression. A higher precedence binds more tightly. When operators have equal precedence, the fixity determines the priority.

When an operator is used in infix position *both* arguments have to be present. Operators can be used in a curried way, but then they have to be used as ordinary prefix functions.

Operator definition.

```
(++) infixr 0:: [a] [a] -> [a]
(++) []      ly  = ly
(++) [x:xs]  ly  = [x:xs ++ ly]

(o) infixr 9:: (a -> b) (c -> a) -> (c -> b)
(o) f g = \x = f (g x)
```

3.2 Patterns

A *pattern* specified on the left-hand side of a function definition specifies the formal arguments of a function. A function alternative is chosen only if the actual arguments of the function application match the formal arguments. A formal argument is either a constant (some *data constructor* with its optional arguments that can consist of sub-patterns) or it is a variable.

```
Pattern          =  [Variable =:] BrackPattern
BrackPattern     =  (GraphPattern)
                 |  QConstructor
                 |  PatternVariable
                 |  SpecialPattern
                 |  DynamicPattern                  // see Chapter 8

GraphPattern     =  QConstructor {Pattern}          // Ordinary data constructor
                 |  GraphPattern QConstructorName   // Infix data constructor
                    GraphPattern
                 |  Pattern

PatternVariable  =  Variable
                 |  _
```

A *pattern variable* can be a (node) *variable* or a *wildcard*. A *variable* is a formal argument of a function that matches on *any* concrete value of the corresponding actual argument and therefore it does *not* force evaluation of this argument. A *wildcard* is an *anonymous* variable ("_") one can use to indicate that the corresponding argument is not used in the right-hand side of the function. A *variable* can be attached to a pattern (using the symbol '`=:`') that makes it possible to identify (*label*) the whole pattern as well as its contents. When a constant (data constructor) is specified as formal argument, the actual argument must contain the same constant in order to have a successful match.

Example of an algebraic data type definition and its use in a pattern match in a function definition.

```
::Tree a = Node a (Tree a) (Tree a)
         | Nil

Mirror:: (Tree a) -> Tree a
Mirror (Node e left right) = Node e (Mirror right) (Mirror left)
Mirror Nil                 = Nil
```

Use of anonymous variables.

```
:: Complex :== (!Real,!Real)                          // synonym type def

realpart:: Complex -> Real
realpart (re,_) = re                                  // re and _ are pattern variables
```

Use of list patterns, use of guards, use of variables to identify patterns and sub-patterns; merge merges two (sorted) lazy lists into one (sorted) list.

```
merge:: [Int] [Int] -> [Int]
merge f []    = f
merge [] s    = s
merge f=:[x:xs] s=:[y:ys]
| x<y         = [x:merge xs s]
| x==y        = merge f ys
| otherwise   = [y:merge f ys]
```

- It is possible that the specified patterns turn a function into a partial function (see 3.7.3). When a partial function is applied outside the domain for which the function is defined it will result into a *run-time* error. A compile time *warning* is generated that such a situation might arise.

The formal arguments of a function and the function body are contained in a new local scope.

```
functionName args = expression
```

- All variable symbols introduced at the left-hand side of a function definition must have different names.

For convenience and efficiency special syntax is provided to express pattern match on data structures of predefined type and record type. They are treated elsewhere (see below).

SpecialPattern	=	BasicValuePattern	// see 4.1.2
	|	ListPattern	// see 4.2.2
	|	TuplePattern	// see 4.3.2
	|	ArrayPattern	// see 4.4.2
	|	RecordPattern	// see 5.2.2
	|	UnitPattern	// see 4.8

3.3 Guards

Guard	=	BooleanExpr
	|	**otherwise**

A *guard* is a Boolean expression attached to a rule alternative that can be regarded as generalisation of the pattern matching mechanism: the alternative only matches when the patterns defined on the left hand-side match *and* its (optional) guard evaluates to `True` (see 3.1). Otherwise the *next* alternative of the function is tried. Pattern matching always takes place *before* the guards are evaluated.

The guards are tried in *textual order*. The alternative corresponding to the first guard that yields `True` will be evaluated. A right-hand side without a guard can be regarded to have a guard that always evaluates to `True`(the 'otherwise' or 'default' case). In `keyword` **`otherwise`** is synonym for `True` for people who like to emphasize the default option.

- Only the last rule alternative of a function can have otherwise as guard or can have no guard.
- It is possible that the guards turn the function into a partial function (see 3.7.3). When a partial function is applied outside the domain for which the function is defined it will result into a *run-time* error. At compile time this cannot be detected.

Function definition with guards.

```
filter:: Int [Int] -> [Int]
filter pr [n:str]
| n mod pr == 0    = filter pr str
| otherwise        = [n:filter pr str]
```

Equivalent definition of previous `filter`.

```
filter:: Int [Int] -> [Int]
filter pr [n:str]
| n mod pr == 0    = filter pr str
                   = [n:filter pr str]
```

Guards can be nested. When a guard on one level evaluates to `True`, the guards on a next level are tried.

- To ensure that at least one of the alternatives of a nested guard will be successful, a nested guarded alternative must always have a 'default case' as last alternative.

Example of a nested guard.

```
example arg1 arg2
| predicate11 arg1                                  // if predicate11 arg1
    | predicate21 arg2 = calculate1 arg1 arg2       //  then (if predicate21 arg2
    | predicate22 arg2 = calculate2 arg1 arg2       //  elseif predicate22 arg2 then
    | otherwise        = calculate3 arg1 arg2       //  else ...)
| predicate12 arg1     = calculate4 arg1 arg2       // elseif predicate12 arg1 then ...
```

3.4 Expressions

The main body of a function is called the *root expression*. The root expression is a graph expression.

RootExpression	=	GraphExpr

GraphExpr	=	Application	
Application	=	{BrackGraph}+	
	|	GraphExpr Operator GraphExpr	
	|	GenericAppExpr	
BrackGraph	=	GraphVariable	
	|	QConstructor	
	|	QFunction	
	|	(GraphExpr)	
	|	LambdaAbstr	// see 3.4.1
	|	CaseExpr	// see 3.4.2
	|	LetExpression	// see 3.5.1
	|	SpecialExpression	
	|	DynamicExpression	
	|	MatchesPatternExpr	// see 3.4.2

QFunction	=	`QFunctionName`
	|	`(QFunctionName)`
QConstructor	=	`QConstructorName`
	|	`(QConstructorName)`
Operator	=	`QFunctionName`
	|	`QConstructorName`
GraphVariable	=	`Variable`
	|	SelectorVariable

An expression generally expresses an application of a function to its actual arguments or the (automatic) creation of a data structure simply by applying a data constructor to its arguments. Each function or data constructor can be used in a *curried* way and can therefore be applied to any number (zero or more) of arguments. A function will only be rewritten if it is applied to a number of arguments equal to the arity of the function (see 3.1). Function and constructors applied on zero arguments just form a syntactic unit (for non-operators no brackets are needed in this case).

- All expressions have to be of correct type (see Chapter 5).
- All symbols that appear in an expression must have been defined somewhere within the scope in which the expression appears (see 2.1).
- There has to be a definition for each node variable and selector variable within in the scope of the graph expression.

Operators are special functions or data constructors defined with arity two which can be applied in infix position The *precedence* (`0` through `9`) and *fixity* (**`infixl`**eft, **`infixr`**ight, **`infix`**) which can be defined in the type definition of the operators determine the priority of the operator application in an expression. A higher precedence binds more tightly. When operators have equal precedence, the fixity determines the priority. In an expression an ordinary function application has a very high priority (`10`). Only selection of record elements (see 5.2.1) and array elements (see 4.4.1) binds more tightly (`11`). Besides that, due to the priority, brackets can sometimes be omitted; operator applications behave just like other applications.

- It is not allowed to apply operators with equal precedence in an expression in such a way tha their fixity conflict. So, when in `a1 op1 a2 op2 a3` the operators `op1` and `op2` have the same precedence a conflict arises when `op1` is defined as infixr implying that the expression must be read as `a1 op1 (a2 op2 a3)` while `op2` is defined as `infixl` implying that the expression must be read as `(a1 op1 a2) op2 a3`.
- When an operator is used in infix position *both* arguments have to be present. Operators can be used in a *curried* way (applied to less than two arguments), but then they have to be used as ordinary *prefix* functions / constructors. When an operator is used as prefix function c.q. constructor, it has to be surrounded by brackets.

There are two kinds of variables that can appear in a graph expression: *variables* introduced as *formal argument* of a function (see 3.1 and 3.2) and *selector variables* (defined in a *selector* to identify parts of a graph expression, see 3.6).

Example of a cyclic root expression. y is the root expression referring to a cyclic graph. The multiplication operator * is used prefix here in a curried way.

```
ham:: [Int]
ham = y
where y = [1:merge (map ((*) 2) y) (merge (map ((*) 3) y) (map ((*) 5) y))]
```

For convenience and efficiency special syntax is provided to create expressions of data structures of predefined type and of record type that is considered as a special kind of algebraic type. They are treated in elsewhere.

SpecialExpression	=	BasicValue	// see 4.1.1
	|	List	// see 4.2.1
	|	Tuple	// see 4.3.1
	|	Array	// see 4.4.1
	|	ArraySelection	// see 4.4.1
	|	Record	// see 5.2.1
	|	RecordSelection	// see 5.2.1
	|	UnitConstructor	// see 4.8

3.4.1 Lambda Abstraction

Sometimes it can be convenient to define a tiny function in an expression "right on the spot". For this purpose one can use a *lambda abstraction*. An anonymous function is defined which can have several formal arguments that can be patterns as common in ordinary function definitions (see Chapter 3). However, only simple functions can be defined in this way: no rule alternatives, and no local function definitions.

It is also allowed to use the arrow ('`->`') to separate the formal arguments from the function body:

LambdaAbstr	=	**** {Pattern}+ {LambdaGuardAlt} {LetBeforeExpression} LambdaResult
LambdaResult	=	**=** GraphExpr
	|	**->** GraphExpr
	|	**|** Guard LambdaGuardRhs

Example of a Lambda expression.

```
AddTupleList:: [(Int,Int)] -> [Int]
AddTupleList list = map ((x,y) = x+y) list
```

A lambda expression introduces a new scope (see 2.1).

The arguments of the anonymous function being defined have the only a meaning in the corresponding function body.

```
\ arg1 arg2 ... argn = function_body
```

Let-before expressions and guards can be used in lambda abstractions:

```
LambdaGuardAlt      =  {LetBeforeExpression} | BooleanExpr LambdaGuardRhs
LambdaGuardRhs      =  {LambdaGuardAlt} {LetBeforeExpression} LambdaGuardResult
LambdaGuardResult   =  =  GraphExpr
                    |  -> GraphExpr
                    |  | otherwise LambdaGuardRhs
```

3.4.2 Case Expression and Conditional Expression

For programming convenience a *case expression* and *conditional expression* are added.

```
CaseExpr            =  case GraphExpr of
                       { {CaseAltDef}+ }
                    |  if BrackGraph BrackGraph BrackGraph
CaseAltDef          =  {Pattern}
                       {CaseGuardAlt} {LetBeforeExpression} CaseResult
                       [LocalFunctionAltDefs]
CaseResult          =  = [>] FunctionBody
                    |  -> FunctionBody
                    |  | Guard CaseGuardRhs
CaseGuardAlt        =  {LetBeforeExpression} | BooleanExpr CaseGuardRhs
CaseGuardRhs        =  {CaseGuardAlt} {LetBeforeExpression} CaseGuardResult
CaseGuardResult     =  = [>] FunctionBody
                    |  -> FunctionBody
                    |  | otherwise CaseGuardRhs
```

In a *case expression* first the discriminating expression is evaluated after which the case alternatives are tried in textual order. Case alternatives are similar to function alternatives. This is not so strange because a case expression is internally translated to a function definition (see the example below). Each alternative contains a left-hand side pattern (see 3.2) that is optionally followed by a *let-before* (see 3.5.4) and a guard (see 3.3). When a pattern matches and the optional guard evaluates to `True` the corresponding alternative is chosen. A new block structure (scope) is created for each case alternative (see 2.1).

It is also allowed to use the arrow ('->') to separate the case alternatives

The variables defined in the patterns have the only a meaning in the corresponding alternative.

```
case expression of
    pattern1 = alternative1
    pattern2 = alternative2
    ...
    patternn = alternativen
```

- All alternatives in the case expression must be of the same type.
- When none of the patterns matches a *run-time* error is generated.

The case expression

```
h x =    case g x of
         [hd:_]  = hd
         []      = abort "result of call g x in h is empty"
```

is semantically equivalent to:

```
h x = mycase (g x)
where
    mycase  [hd:_]  = hd
    mycase  []      = abort "result of call g x in h is empty"
```

In a *conditional expression* first the Boolean expression is evaluated after which either the then- or the else-part is chosen. The conditional expression can be seen as a simple kind of case expression.

- The then- and else-part in the conditional expression must be of the same type.
- The discriminating expression must be of type `Bool`.

3.4.3 Matches Pattern Expression

`expression =: pattern` in an expression yields `True` if the `expression` matches the `pattern` and `False` otherwise. Variable names are not allowed in the `pattern`, but _'s may be used. The compiler converts the expression to a case if the pattern is more complicated than just a constructor and _'s.

```
MatchesPatternExpr      =   GraphExpr =: QConstructorName { _ }
                        |   GraphExpr =: BrackPattern
```

`=:` in expressions binds stronger than function application. This includes additional _'s after a `ConstructorName`, so `x=:(Constructor _ _)` may be written without parenthesis as `x=:Constructor _ _`, for example:

```
:: T = X Int | Y Int Int | Z;   // algebraic type definition

isXorY t = t=:X _ || t=:Y _ _;
```

3.5 Local Definitions

Sometimes it is convenient to introduce definitions that have a limited scope and are not visible throughout the whole module. One can define *functions* that have a local scope, i.e. which have only a meaning in a certain program region.

Outside the scope the functions are unknown. This locality can be used to get a better program structure: functions that are only used in a certain program area can remain hidden outside that area.

Besides functions one can also define constant selectors. Constants are named graph expressions (see 3.6).

```
LocalDef                =   GraphDef
                        |   FunctionDef
```

3.5.1 Let Expression: Local Definitions in Expressions

A *let* expression is an expression that enables to introduce a new scope (see 2.1) in an expression in which local functions and constants can be defined. Such local definitions can be introduced anywhere in an expression using a let expression with the following syntax.

```
LetExpression           =   let { {LocalDef}+ } in GraphExpr
```

The function and selectors defined in the let block only have a meaning within the *expression*.

```
let
    function arguments = function_body
    selector = expr
    ...
in  expression
```

Example of a let expression used within a list comprehension.

```
doublefibs n = [let a = fib i in (a, a) \\ i <- [0..n]]
```

3.5.2 Where Block: Local Definitions in a Function Alternative

At the end of each function alternative one can locally define functions and constant graphs in a *where block*.

```
LocalFunctionAltDefs    =   [where] { {LocalDef}+ }
```

Functions and graphs defined in a *where* block can be used anywhere in the corresponding function alternative (i.e. in all guards and rule alternatives following a pattern, see 3.1) as indicated in the following picture showing the scope of a *where* block.

The function and selectors defined in the where block can be used locally in the whole function definition.

```
function formal_arguments
         | guard1     = function_alternative1
         | guard2     = function_alternative2
         | otherwise  = default_alternative
         where
             selector = expr
             local_function args = function_body
```

sieve and filter are local functions defined in a where block. They have only a meaning inside primes. At the global level the functions are unknown.

```
primes::[Int]
primes = sieve [2..]
where
    sieve::[Int] -> [Int]                          // local function of primes
    sieve [pr:r]  = [pr:sieve (filter pr r)]

    filter::Int [Int] -> [Int]                     // local function of primes
    filter pr [n:r]
    | n mod pr == 0    = filter pr r
    | otherwise        = [n:filter pr r]
```

Notice that the scope rules are such that the formal arguments of the surrounding function alternative are visible to the locally defined functions and graphs. The arguments can therefore directly be addressed in the local definitions. Such local definitions cannot always be typed explicitly (see 3.7).

Alternative definition of `primes`. The function filter is locally defined for `sieve.filter` can directly access arguments `pr` of `sieve`.

```
primes::[Int]
primes = sieve [2..]
where
    sieve::[Int] -> [Int]                          // local function of primes
    sieve [pr:r]  = [pr:sieve (filter pr r)]
    where
        filter::Int [Int] -> [Int]                 // local function of sieve
        filter pr [n:r]
        | n mod pr == 0    = filter pr r
        | otherwise        = [n:filter pr r]
```

3.5.3 With Block: Local Definitions in a Guarded Alternative

One can also locally define functions and graphs at the end of each guarded rule alternative using a *with block*.

LocalFunctionDefs	=	[**with**] **{** {LocalDef}+ **}**
LocalDef	=	GraphDef
	|	FunctionDef

Functions and graphs defined in a *with* block can only be used in the corresponding rule alternative as indicated in the following picture showing the scope of a *with* block.

The function and selectors defined in the *with* block can be locally only be used in the corresponding function alternative.

```
function formal arguments
         | guard1     =   function_alternative1
                         with
                              selector = expr
                              local_function args = function_body

         | guard2     =   function_alternative2
                         with
                              selector = expr
                              local_function args = function_body
```

Notice that the scope rules are such that the arguments of the surrounding guarded rule alternative are visible to the locally defined functions and graphs. The arguments can therefore directly be addressed in the local definitions. Such local definitions cannot always be typed explicitly (see 3.7).

3.5.4 Let-Before Expression: Local Constants defined between Guards

Many of the functions for input and output in the CLEAN I/O library are state transition functions. Such a state is often passed from one function to another in a single threaded way (see Chapter 9) to force a specific order of evaluation. This is certainly the case when the state is of unique type. The threading parameter has to be renamed to distinguish its different versions. The following example shows a typical example:

Use of state transition functions. The uniquely typed state file is passed from one function to another involving a number of renamings: file, file1, file2)

```
readchars:: *File -> ([Char], *File)
readchars file
| not ok     = ([],file1)
| otherwise  = ([char:chars], file2)
where
    (ok,char,file1)    = freadc file
    (chars,file2)      = readchars file1
```

This explicit renaming of threaded parameters not only looks very ugly, these kind of definitions are sometimes also hard to read as well (in which order do things happen? which state is passed in which situation?). We have to admit: an imperative style of programming is much easier to read when things have to happen in a certain order such as is the case when doing I/O. That is why we have introduced *let-before* expressions.

Let-before expressions are special let expressions that can be defined before a guard or function body. In this way one can specify sequential actions in the order in which they suppose to happen. *Let-before* expressions have the following syntax:

```
LetBeforeExpression     =  # {GraphDefOrUpdate}+
                        |  #!{GraphDefOrUpdate}+
GraphDefOrUpdate        =  GraphDef
                        |  Variable & {FieldName {Selection} = GraphExpr}-list ;                  // see 5.2.1
                        |  Variable & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list] ;   // see 4.4.1
```

The form with the exclamation mark (`#!`) forces the evaluation of the node-ids that appear in the left-hand sides of the definitions. Notice that one can only define constant selectors (GraphDef) in a Let-before expression. One cannot define functions.

Let-before expressions have a special scope rule to obtain an imperative programming look. The variables in the left-hand side of these definitions do not appear in the scope of the right-hand side of that definition, but they do appear in the scope of the other definitions that follow (including the root expression, excluding local definitions in where blocks.

This is shown in the following picture:

```
Function args
         # selector1  = expression1
         | guard1     = expression2
         # selector2  = expression3
         | guard2     = expression4
         where
             local_definitions
```

Notice that a variable defined in a let-before expression cannot be used in a where expression. The reverse is true however: definitions in the where expression can be used in the let before expression.

Use of let before expressions, short notation, re-using names taking use of the special scope of the let before)

```
readchars:: *File -> ([Char], *File)
readchars file
#    (ok,char,file)     = freadc file
|    not ok             = ([],file)
#    (chars,file)       = readchars file
=    ([char:chars], file)
```

Equivalent definition renaming threaded parameters

```
readchars:: *File -> ([Char], *File)
readchars file
#    (ok,char,file1)    = freadc file
|    not ok             = ([],file1)
#    (chars, file2)     = readchars file1
=    ([char:chars], file2)
```

The notation can also be dangerous: the same name is used on different spots while the meaning of the name is not always the same (one has to take the scope into account which changes from definition to definition). However, the notation is rather safe when it is used to thread parameters of unique type. The type system will spot it when such parameters are not used in a correct single threaded manner. We do not recommend the use of let before expressions to adopt an imperative programming style for other cases.

Abuse of let before expression.

```
exchange:: (a, b) -> (b, a)
exchange (x, y)
#    temp = x
     x    = y
     y    = temp
=    (x, y)
```

3.6 Defining Constants

One can give a name to a constant expression (actually a graph), such that the expression can be used in (and shared by) other expressions. One can also identify certain parts of a constant via a projection function called a selector (see below).

GraphDef = Selector =[:] GraphExpr ;

Graph locally defined in a function: the graph labeled `last` is shared in the function `StripNewline` and computed only once.

```
StripNewline:: String -> String
StripNewline "" = ""
StripNewline string
| string !! last<>'\n' = string
| otherwise            = string%(0,last-1)
where
    last = maxindex string
```

When a *graph* is *defined* actually a name is given to (part) of an expression. The definition of a graph can be compared with a definition of a *constant*(data) or a*constant* (*projection*) *function*. However, notice that graphs are constructed according to the basic semantics of CLEAN (see Chapter 1) that means that multiple references to the same graph will result in *sharing* of that graph. Recursive references will result in *cyclic graph structures*. Graphs have the property that they *are computed only once* and that their value is *remembered*within the scope they are defined in.

Graph definitions differ from constant function definitions. A *constant function definition* is just a function defined with arity zero (see 3.1). A constant function defines an ordinary graph rewriting rule: multiple references to a function just means that the same definition is used such that a (constant) function *will be recomputed again* for each occurrence of *the function symbol made*. This difference can have consequences for the time and space behavior of function definitions (see 10.2).

The Hamming numbers defined using a locally defined cyclic constant graph and defined by using a globally defined recursive constant function. The first definition (ham1) is efficient because already computed numbers are reused via sharing. The second definition (ham2) is much more inefficient because the recursive function recomputes everything.

```
ham1:: [Int]
ham1 = y
where y = [1:merge (map ((*) 2) y) (merge (map ((*) 3) y) (map ((*) 5) y))]

ham2:: [Int]
ham2 = [1:merge (map ((*) 2) ham2) (merge (map ((*) 3) ham2) (map ((*) 5) ham2 ))]
```

Syntactically the definition of a graph is distinguished from the definition of a function by the symbol which separates the left-hand side from the right-hand side: "=" or "=>" is used for functions, while "=" is used for local graphs and "=:" for global graphs. However, in general "=" is used both for functions and local graphs. Generally it is clear from the context which is meant (functions have parameters, selectors are also easy recognisible). However, when a simple constant is defined the syntax is ambiguous (it can be a constant function definition as well as a constant graph definition).

To allow the use of the "=" whenever possible, the following rule is followed. Local constant definitions are *by default* taken to be *graph* definitions and therefore shared, globally they are *by default* taken to be *function* definitions (see 3.1) and therefore recomputed. If one wants to obtain a different behavior one has to explicit state the nature of the constant definition (has it to be shared or has it to be recomputed) by using "=:" (on the global level, meaning it is a constant graph which is shared) or "=>" (on the local level, meaning it is a constant function and has to be recomputed).

Local constant graph versus local constant function definition: `biglist1` and `biglist2` is a *graph* which is computed only once, `biglist3` is a constant *function* which is computed every time it is applied.

```
biglist1 =   [1..10000]                // a graph (if defined locally)
biglist1 =   [1..10000]                // a constant function (if defined globally)
biglist2 =:  [1..10000]                // a graph (if defined globally)
biglist3 =>  [1..10000]                // a constant function (always)
```

The garbage collector will collect locally defined graphs when they are no longer connected to the root of the program graph (see Chapter 1).

3.6.1 Selectors

The left-hand side of a graph definition can be a simple name, but is can also be a more complicated pattern called a selector. A *selector* is a pattern which introduces one or more new *selector variables* implicitly defining *projection functions* to identify (parts of) a constant graph being defined One can identify the sub-graph as a whole or one can identify its components. A selector can contain constants (also user defined constants introduced by algebraic type definitions), variables and wildcards. With a *wildcard* one can indicate that one is not interested in certain components.

Selectors cannot be defined globally. They can only locally be defined in a let (see 3.5.1), a let-before (see 3.5.4), a where-block (see 3.5.2), and a with-block (see 3.5.3). Selectors can furthermore appear on the left-hand side of generators in list comprehensions (see 4.2.1) and array comprehensions (see 4.4.1).

Selector	=	BrackPattern	// for bracket patterns see 3.2

Use of a selector to locally select tuple elements.

```
unzip::[(a,b)] -> ([a],[b])
unzip []          = ([],[])
unzip [(x,y):xys] = ([x:xs],[y:ys])
where
    (xs,ys) = unzip xys
```

- When a selector on the left-hand side of a graph definition is not matching the graph on the right-hand side it will result in a *run-time* error.
- The selector variables introduced in the selector must be different from each other and not already be used in the same scope and name space (see 1.2).
- To avoid the specification of patterns that may fail at run-time, it is not allowed to test on zero arity constructors. For instance, list used in a selector pattern need to be of form `[a:_]`. `[a]` cannot be used because it stands for `[a:[]]` implying a test on the zero arity constructor `[]`. If the pattern is a record only those fields which contents one is interested in need to be indicated in the pattern
- Arrays cannot be used as pattern in a selector.
- Selectors cannot be defined globally.

3.7 Typing Functions

Although one is in general not obligated to explicitly specify the type of a function (the CLEAN compiler can in general `infer` the type) the explicit specification of the type is `highly recommended` to increase the readability of the program.

```
FunctionDef          =  [FunctionTypeDef]
                        DefOfFunction

FunctionTypeDef      =  FunctionName :: FunctionType ;
                     |  (FunctionName) [FixPrec] [:: FunctionType] ;
FixPrec              =  infixl [Prec]
                     |  infixr [Prec]
                     |  infix [Prec]
Prec                 =  Digit
FunctionType         =  [{ArgType}+ ->] Type [ClassContext] [UnqTypeUnEqualities]
ArgType              =  BrackType
                     |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type [ClassContext])
Type                 =  {BrackType}+
BrackType            =  [Strict] [UnqTypeAttrib] SimpleType
UnivQuantVariables   =  A.{TypeVariable }+:
```

An explicit specification is `required` when a function is exported, or when the programmer wants to impose additional restrictions on the application of the function (e.g. a more restricted type can be specified, strictness information can be added as explained in Chapter 10.1, a class context for the type variables to express overloading can be defined as explained in Chapter 7, uniqueness information can be added as explained in 3.7.5 Functions with Strict Arguments).

The CLEAN type system uses a combination of Milner/Mycroft type assignment. This has as consequence that the type system in some rare cases is not capable to infer the type of a function (using the Milner/Hindley system) although it will approve a given type (using the Mycroft system; see Plasmeijer and Van Eekelen, 1993). Also when universally quantified types of rank 2 are used (see 3.7.4), explicit typing by the programmer is required.

The Cartesian product is used for the specification of the function type. The Cartesian product is denoted by juxtaposition of the bracketed argument types. For the case of a single argument the brackets can be left out. In type specifications the binding priority of the application of type constructors is higher than the binding of the arrow `->`. To indicate that one defines an operator the function name is on the left-hand side surrounded by brackets.

- The function symbol before the double colon should be the same as the function symbol of the corresponding rewrite rule.
- The arity of the functions has to correspond with the number of arguments of which the Cartesian product is taken. So, in CLEAN one can tell the arity of the function by its type.

Showing how the arity of a function is reflected in type.

```
map:: (a->b) [a] -> [b]                                  // map has arity 2
map f []     =   []
map f [x:xs] =   [f x : map f xs]

domap:: ((a->b) [a] -> [b])                              // domap has arity zero
domap = map
```

- The arguments and the result types of a function should be of kind `X`.
- In the specification of a type of a locally defined function one cannot refer to a type variable introduced in the type specification of a surrounding function (there is not yet a scope rule on types defined). The programmer can therefore not specify the type of such a local function. However, the type will be inferred and checked (after it is lifted by the compiler to the global level) by the type system.

Counter example (illegal type specification). The function `g` returns a tuple. The type of the first tuple element is the same as the type of the polymorphic argument of `f`. Such a dependency (here indicated by `"^" cannot be specified`) **.**

```
f:: a -> (a,a)
f x = g x
where
    // g:: b -> (a^,b)
    g y = (x,y)
```

3.7.1 Typing Curried Functions

In CLEAN all symbols (functions and constructors) are defined with *fixed arity*. However, in an application it is of course allowed to apply them to an arbitrary number of arguments. A *curried application* of a function is an application of a function with a number of arguments which is less than its arity (note that in CLEAN the arity of a function can be derived from its type). With the aid of the predefined internal function *_AP* a curried function applied on the required number of arguments is transformed into an equivalent uncurried function application.

The type axiom's of the CLEAN type system include for all `s` defined with arity `n` the equivalence of `s::(t1->(t2->(...(tn->tr)...))` with `s::t1 t2 ... tn -> tr`.

3.7.2 Typing Operators

An *operator* is a *function with arity two* that can be used in infix position. An operator can be defined by enclosing the operator name between parentheses in the left-hand-side of the function definition. An operator has a *precedence* (`0` through `9`, default `9`) and a **`fixity`** (**`infixl`**, **`infixr`** or just **`infix`**, default **`infixl`**). A higher precedence binds more tightly. When operators have equal precedence, the fixity determines the priority. In an expression an ordinary function application always has the highest priority (`10`). See also Section 3.1 and 3.4.

- The type of an operator must obey the requirements as defined for typing functions with arity two.
- If the operator is explicitly typed the operator name should also be put between parentheses in the type rule.
- When an infix operator is enclosed between parentheses it can be applied as a prefix function. Possible recursive definitions of the newly defined operator on the right-hand-side also follow this convention.

Example of an operator definition and its type.

```
(o) infix 8:: (x -> y) (z -> x) -> (z -> y)                // function composition
(o) f g = \x -> f (g x)
```

3.7.3 Typing Partial Functions

Patterns and guards imply a condition that has to be fulfilled before a rewrite rule can be applied (see 3.2 and 3.3). This makes it possible to define *partial function s*, functions which are not defined for all possible values of the specified type.

- When a partial function is applied to a value outside the domain for which the function is defined it will result into a *run-time* error. The compiler gives a warning when functions are defined which might be partial.

With the `abort` expression (see `StdMisc.dcl`) one can change any partial function into a *total function* (the `abort` expression can have any type). The abort expression can be used to give a user-defined run-time error message

Use of abort to make a function total.

```
fac:: Int -> Int
fac 0         = 1
fac n
| n>=1        = n * fac (n - 1)
| otherwise   = abort "fac called with a negative number"
```

3.7.4 Explicit use of the Universal Quantifier in Function Types

When a type of a polymorphic function is specified in CLEAN, the universal quantifier is generally left out.

The function `map` defined as usual, no universal quantifier is specified:

```
map:: (a->b) [a] -> [b]
map f []     =   []
map f [x:xs] =   [f x : map f xs]
```

Counter Example. The same function `map` again, but now the implicit assumed universal quantifier has been made visible. It shows the meaning of the specified type more precisely, but it makes the type definition a bit longer as well. **The current version of Clean does not yet allow universal quantifiers on the topmost level !!**

```
map:: A.a b: (a->b) [a] -> [b]
map f []     =   []
map f [x:xs] =   [f x : map f xs]
```

Not yet Implemented: In Clean 2.0 it is allowed to explicitly write down the universal quantifier. One can write down the qualifier A. (for all) direct after the :: in the type definition of a function. In this way one can explicitly introduce the type variables used in the type definition of the function. As usual, the type variables thus introduced have the whole function type definition as scope.

FunctionType	=	[{ArgType}+ **->**] Type [ClassContext] [UnqTypeUnEqualities]
ArgType	=	BrackType
	|	[Strict] [UnqTypeAttrib] **(**UnivQuantVariables Type [ClassContext]**)**
Type	=	{BrackType}+
BrackType	=	[Strict] [UnqTypeAttrib] SimpleType
UnivQuantVariables	=	**A.**{TypeVariable }+**:**

CLEAN offers Rank 2 polymorphism: it is possible to specify the universal quantifier with as scope the type of an argument of a function. This makes it possible to pass polymorphic functions as an argument to a function which otherwise would be treated monomorphic. The advantage of the use of Rank 2 polymorphism is that more programs will be approved by the type system, but one explicitly (by writing down the universal quantifier) has to specify in the type of function that such a polymorphic function is expected as argument or delivered as result.

Example: The function `h` is used to apply a polymorphic function of type `(A.a: [a] -> Int)` to a list of `Int`as well as a list of `Char`. Due to the explicit use of the universal quantifier in the type specification of `h` this definition is approved.

```
h:: (A.a: [a] -> Int) -> Int
h f = f [1..100] + f ['a'..'z']

Start = h length
```

Counter Example: The function `h2` is used to apply a function of type `([a] -> Int)` to a list of Int as well as a list of `Char` . In this case the definition is rejected due to a type unification error. It is assumed that the argument of `h2` is unifiable with `[a] -> Int`, but it is not assumed that the argument of `h2` is `(A.a: [a] -> Int)`. So, the type variable a is unified with both `Int` and `Char`, which gives rise to a type error.

```
h2:: ([a] -> Int) -> Int
h2 f = f [1..100] + f ['a'..'z']

Start = h2 length
```

Counter Example: The function h3 is used to apply a function to a list of `Int` as well as a list of `Char`. Since no type is specified the type inference system will assume `f` to be of type `([a] -> Int)` but not of type `(A.a: [a] -> Int)`. The situation is the same as above and we will again get a type error.

```
h3 f = f [1..100] + f ['a'..'z']

Start = h3 length
```

- CLEAN cannot infer polymorphic functions of Rank 2 automatically! One is obligated to explicitly specify universally quantified types of Rank 2.
- Explicit universal quantification on higher ranks than rank 2 (e.g. quantifiers specified somewhere inside the type specification of a function argument) is not allowed.
- A polymorphic function of Rank 2 cannot be used in a curried way for those arguments in which the function is universally quantified.

Counter Examples: In the example below it is shown that `f1` can only be used when applied to *all* its arguments since its last argument is universally quantified. The function `f2` can be used curried only with respect to its last argument that is not universally quantified.

```
f1:: a (A.b:b->b) -> a
f1 x id = id x

f2:: (A.b:b->b) a -> a
f2 id x = id x

illegal1 = f1                                   // this will raise a type error

illegal2 = f1 3                                 // this will raise a type error

legal1 :: Int
legal1 = f1 3 id where id x = x                 // ok

illegal3 = f2                                   // this will raise a type error

legal2 :: (a -> a)
legal2 = f2 id where id x = x                   // ok

legal3 :: Int
legal3 = f2 id 3 where id x = x                 // ok
```

3.7.5 Functions with Strict Arguments

In the type definition of a function the arguments can optionally be annotated as being strict. In reasoning about functions it will always be true that the corresponding arguments will be in strong root normal form (see 2.1) before the rewriting of the function takes place. In general, strictness information will increase the efficiency of execution (see Chapter 10).

Example of a function with strict annotated arguments.

```
Acker:: !Int !Int -> Int
Acker 0 j =  inc j
Acker i 0 =  Acker (dec i) 1
Acker i j =  Acker (dec i) (Acker i (dec j))
```

The CLEAN compiler includes a fast and clever strictness analyzer that is based on abstract reduction (Nöcker, 1993). The compiler can derive the strictness of the function arguments in many cases, such as for the example above. Therefore there is generally no need to add strictness annotations to the type of a function by hand. When a function is exported from a module (see Chapter 2), its type has to be specified in the definition module. To obtain optimal efficiency, the programmer should also include the strictness information to the type definition in the definition module. One can ask the compiler to print out the types with the derived strictness information and paste this into the definition module.

Notice that strictness annotations are only allowed at the outermost level of the argument type. Strictness annotations inside type instances of arguments are not possible (with exception for some predefined types like tuples and lists). Any (part of) a data structure can be changed from lazy to strict, but this has to be specified in the type definition (see 5.1.6).

Chapter 4

Predefined Types

Certain types like `Int`egers, `Bool`eans, `Char`acters, `Real`s, Lists, Tuples and Arrays are that frequently used that they have been predefined in CLEAN for reasons of efficiency and/or notational convenience. These types and the syntactic sugar that has been added to create and to inspect (via pattern matching) objects of these popular types are treated in this chapter.

```
PredefinedType  =  BasicType      // see 4.1
                |  ListType       // see 4.2
                |  TupleType      // see 4.3
                |  ArrayType      // see 4.4
                |  ArrowType      // see 4.6
                |  PredefType     // see 4.7
```

In Chapter 5 we will explain how new types can be defined.

4.1 Basic Types: Int, Real, Char and Bool

Basic types are *algebraic types* (see 5.1) which are predefined for reasons of efficiency and convenience: `Int` (for 32 or 64 bits integer values), `Real` (for 64 bit double precision floating point values), `Char` (for 8 bits ASCII character values) and `Bool` (for 8 bits Boolean values). For programming convenience special syntax is introduced to denote constant values (data constructors) of these predefined types. Functions to create and manipulate objects of basic types can be found in the CLEAN `StdEnv` library (as indicated below).

There is also a special notation to denote a string (an unboxed array of characters, see 4.4) as well as to denote a list of characters (see 4.2.1).

```
BasicType  =  Int          // see StdInt.dcl
           |  Real         // see StdReal.dcl
           |  Char         // see StdChar.dcl
           |  Bool         // see StdBool.dcl
```

4.1.1 Creating Constant Values of Basic Type

In a graph expression a *constant value*of basic type `Int`, `Real`, `Bool` or `Char` can be created.

```
BasicValue  =  IntDenotation
            |  RealDenotation
            |  BoolDenotation
            |  CharDenotation
```

```
IntDenotation   =  [Sign]{Digit}+              // decimal number
                |  [Sign]0{OctDigit}+          // octal number
                |  [Sign]0x{HexDigit}+         // hexadecimal number
Sign            =  + | -
RealDenotation  =  [Sign]{Digit}+.{Digit}+[E[Sign]{Digit}+]
BoolDenotation  =  True | False
CharDenotation  =  CharDel AnyChar/CharDel CharDel
CharsDenotation =  CharDel {AnyChar/CharDel}+ CharDel
```

```
AnyChar       = IdChar | ReservedChar | SpecialChar
ReservedChar  = (  |  )  |  {  |  }  |  [  |  ]  |  ;  |  ,  |  .
SpecialChar   = \n  |  \r  |  \f  |  \b          // newline,return,formf,backspace
              |  \t  |  \\  |  \CharDel          // tab,backslash,character delimiter
              |  \StringDel                      // string delimiter
              |  \{OctDigit}+                    // octal number
              |  \x{HexDigit}+                   // hexadecimal number
```

```
Digit     = 0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
OctDigit  = 0  |  1  |  2  |  3  |  4  |  5  |  6  |  7
HexDigit  = 0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
          |  A  |  B  |  C  |  D  |  E  |  F
          |  a  |  b  |  c  |  d  |  e  |  f
```

```
CharDel   = '
StringDel = "
```

Example of denotations.

```
Integer (decimal):      0|1|2|...|8|9|10| ... |-1|-2| ...
Integer (octal):        00|01|02|...|07|010| ... |-01|-02| ...
Integer (hexadecimal):  0x0|0x1|0x2|...|0x8|0x9|0xA|0xB ... |-0x1|-0x2| ...
Real:                   0.0|1.5|0.314E10| ...
Boolean:                True | False
Character:              'a'|'b'|...|'A'|'B'|...
String:                 "" | "Rinus"|"Marko"|...
List of characters:     ['Rinus']|['Marko']|...
```

4.1.2 Patterns of Basic Type

A *constant value* of predefined *basic type* `Int`, `Real`, `Bool` or `Char` (see 4.1) can be specified as pattern.

- The denotation of such a value must obey the syntactic description given in above.

Use of Integer values in a pattern.

```
nfib:: Int -> Int
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + nfib (n-1) * nfib (n-2)
```

4.2 Lists

A *list*is an algebraic data type predefined just for programming convenience. A list can contain an *infinite number* of elements. All elements must be of the *same type*. Lists are very often used in functional languages and therefore the usual syntactic sugar is provided for the creation and manipulation of lists (dot-dot expressions, list comprehensions) while there is also special syntax for a *list of characters*.

Lists can be lazy (default), and optionally be defined as head strict, spine strict, strict (both head and spine strict), head strict unboxed, and strict unboxed. Lazy, strict and unboxed lists are all objects of *different:* type. All these different types of lists have different time and space properties (see 10.1.3). Because these lists are of different type, conversion functions are needed to change e.g. a lazy list to a strict list. Functions defined on one type of a list cannot be applied to another type of list. However, one can define overloaded functions that can be used on any list: lazy, strict as well as on unboxed lists.

```
ListType       = [[ListTypeKind] Type [SpineStrictness]]
ListTypeKind   = !                          // head strict list
               | #                          // head strict, unboxed list
SpineStrictness = !                         // tail (spine) strict list
```

- All elements of a list must be of the *same type*.

4.2.1 Creating Lists

Because lists are very convenient and very frequently used data structures, there are several syntactical constructs in CLEAN for creating lists, including *dot-dot expressions* and *ZF-expressions*. Since CLEAN is a lazy functional language, the default list in CLEAN is a *lazy* list. However, in some cases *strict lists*, *spine strict lists* and *unboxed* lists can be more convenient and more efficient.

```
List                    =  ListDenotation
                        |  DotDotExpression
                        |  ZF-expression
```

- All elements of a list must be of the same type.

Lazy Lists

```
ListDenotation          =  [[ListKind] [{LGraphExpr}-list [: GraphExpr]] [SpineStrictness] ]
LGraphExpr              =  GraphExpr
                        |  CharsDenotation
```

```
CharsDenotation         =  CharDel {AnyChar/CharDel}+ CharDel
CharDel                 =  '
```

One way to create a list is by explicit enumeration of the list elements. Lists are constructed by adding one or more elements to an existing list.

Various ways to define a lazy list with the integer elements `1,3,5,7,9`.

```
[1:[3:[5:[7:[9:[]]]]]]
[1:3:5:7:9:[]]
[1,3,5,7,9]
[1:[3,5,7,9]]
[1,3,5:[7,9]]
```

A special notation is provided for the frequently used *list of characters*.

Various ways to define a lazy list with the characters '`a`', '`b`' and '`c`'.

```
['a':['b':['c':[]]]]
['a','b','c']
['abc']
['ab','c']
```

Strict , Unboxed and Overloaded Lists

```
ListKind                =  !                    // head strict list
                        |  #                    // unboxed list
                        |  |                    // overloaded list
SpineStrictness         =  !                    // spine strict list
```

In CLEAN *any* data structure can be made (partially) strict or unboxed (see 10.1). This has consequences for the time and space behavior of the data structure.

For instance, *lazy lists* are very convenient (nothing is evaluated unless it is really needed for the computation, one can deal with infinite data structures), but they can be inefficient as well if actually always all lists elements are evaluated sooner or later. *Strict lists* are often more efficient, but one has to be certain not to trigger a not used infinite computation. *Spine strict lists* can be more efficient as well, but one cannot handle infinite lists in this way. Unboxed lists are head strict. The difference with a strict list is that the representation of an *unboxed list* is more compact: instead of a pointer to the lists element the list element itself is stored in the list. However, unboxed lists have as disadvantage that they can only be used in certain cases: they can only contain elements of basic type, records and tuples. It does not make sense to offer unboxing for arbitrary types: boxing saves space, but not if Cons nodes are copied often: the lists elements are copied as well while otherwise the contents could remain shared using the element pointer instead.

In terms of efficiency it can make quite a difference (e.g. strict lists can sometimes be 6 times faster) which kind of list is actually used. But, it is in general not decidable which kind of list is best to use. This depends on how a list is used in a program. A wrong choice might turn a program from useful to useless (too inefficient), from terminating to non-terminating.

Because lists are so frequently used, special syntax is provided to make it easier for a programmer to change from one type of list to another, just by changing the kind of brackets used. One can define a list of which the *head*element is strict but the spine is lazy (indicated by `[! ]`), a list of which the *spine* is strict but the head element is lazy (indicated by `[ !]`) and a completely evaluated list (indicated by `[! !]`). One can have an unboxed list with a strict head element (indicated by `[# ]`) and a completely evaluated unboxed list of which in addition the spine is strict as well (indicated by *[# !]*

Note that all these different lists are of different type and consequently these lists cannot be mixed and unified with each other. With conversion functions offered in the CLEAN libraries it is possible to convert one list type into another. It is also possible to define an overloaded list and overloaded functions that work on *any* list (see hereafter).

Various types of lists.

```
[  fac 10 : expression  ]                  // lazy list
[! fac 10 : expression  ]                  // head strict list
[! fac 10 : expression !]                  // head strict and tail strict list
[# fac 10 : expression  ]                  // head strict list, unboxed
[# fac 10 : expression !]                  // head strict and tail strict list, unboxed
```

Unboxed data structures can only contain elements of basic type, records and arrays.

One can create an *overloaded* list that will fit on any type of list (lazy, strict or unboxed)

Example of an overloaded list.

```
[| fac 10 : expression ]                   // overloaded list
```

Other ways to create lists are via dot-dot expressions and list comprehensions.

DotDot Expressions

DotDotExpression = [[ListKind] GraphExpr [, GraphExpr] . . [GraphExpr] [SpineStrictness]]

With a dot-dot expression the list elements can be enumerated by giving the first element (`n1`), an optional second element (`n2`) and an optional last element (`e`).

Alternative ways to define a list a dot dot expression.

```
[1,3..9]                                   // [1,3,5,7,9]
[1..9]                                     // [1,2,3,4,5,6,7,8,9]
[1..]                                      // [1,2,3,4,5 and so on ...
['a'..'c']                                 // ['abc']
```

The generated list is in general calculated as follows:

```
from_then_to:: !a !a !a -> .[a] | Enum a
from_then_to n1 n2 e
| n1 <= n2   = _from_by_to n1 (n2-n1) e
             = _from_by_down_to n1 (n2-n1) e
where
    from_by_to n s e
    | n<=e   = [n : _from_by_to (n+s) s e]
             = []

    from_by_down_to n s e
    | n>=e   = [n : _from_by_down_to (n+s) s e]
             = []
```

The step size is *one* by default. If no last element is specified an infinite list is generated.

With a dot-dot expression one can also define a lazy list, a strict list, an unboxed list or an overloaded list.

Different types of lists (lazy, strict, unboxed and overloaded can be defined with a dot dot expression as well.

```
[  1,3..9  ]                          // a lazy list
[! 1,3..9  ]                          // a head strict list
[! 1,3..9 !]                          // a strict list (head and spine)
[# 1,3..9  ]                          // a head strict list, unboxed
[# 1,3..9 !]                          // a strict list (head and spine), unboxed
[| 1,3..9  ]                          // an overloaded list
```

- Dot-dot expression can only be used if one imports `StdEnum from the standard library.`
- Dot-dot expressions are predefined on objects of type `Int`, `Real` and `Char`, but dot-dots can also be applied to any user defined data structure for which the class enumeration type has been instantiated (see CLEANs Standard Environment).

List Comprehensions

```
ZF-expression   =  [[ListKind] GraphExpr \\ {Qualifier}-list [SpineStrictness]]
Qualifier       =  Generators {, let { {LocalDef}+ } } {|Guard}
Generators      =  Generator {& Generator}
Generator       =  Selector <- ListExpr              // select from a lazy list
                |  Selector <|- ListExpr             // select from an overloaded list
                |  Selector <-: ArrayExpr            // select from an array
Selector        =  BrackPattern                      // for brack patterns see 3.2
ListExpr        =  GraphExpr
ArrayExpr       =  GraphExpr
Guard           =  BooleanExpr
BooleanExpr     =  GraphExpr
ListKind        =  !                                 // head strict list
                |  #                                 // unboxed list
                |  |                                 // overloaded list
SpineStrictness =  !                                 // spine strict list
```

With a list generator called a ZF-expression one can construct a list composed from elements drawn from other lists or arrays. With a *list generator*one can draw elements from lazy list. The symbol '`<-`' is used to draw elements from a lazy list, the symbol '`<|-`' can be used to draw elements from any (lazy, strict, unboxed) list. With an *array generator* (use symbol '`<-:`') one can draw elements from any array. One can define several generators in a row separated by a comma. The last generator in such a sequence will vary first. One can also define several generators in a row separated by a '`&`'. All generators in such a sequence will vary at the same time but the drawing of elements will stop as soon of one the generators is exhausted. This construct can be used instead of the zip-functions that are commonly used. *Selectors* are simple patterns to identify parts of a graph expression. They are explained in Section 3.6. Only those lists produced by a generator that match the specified selector are taken into account. Guards can be used as filter in the usual way.

The scope of the selector variables introduced on the left-hand side of a generator is such that the variables can be used in the guards and other generators that follow. All variables introduced in this way can be used in the expression before the `\\` (see the picture below).

```
[ expression \\  selector  <- expression
            |   guard
            ,   selector  <- expression
            |   guard
]
```

ZF-expression:
expr1 yields [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2)]
expr2 yields [(0,0), (1,1), (2,2)].
expr3 yields [(0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2), (3,3)])

```
expr1 = [(x,y) \\ x <- [0..3] , y <- [0..2]]
expr2 = [(x,y) \\ x <- [0..3] & y <- [0..2]]
expr3 = [(x,y) \\ x <- [0..3] , y <- [0..x]]
```

ZF-expression: a well-know sort.

```
sort:: [a] -> [a] | Ord a
sort []      = []
sort [p:ps]  = sort [x \\ x<-ps | x<=p] ++ [p] ++ sort [x \\ x<-ps | x>p]
```

ZF-expression sorting a strict (head and tail strict) list.

```
ssort:: [!a!] -> [!a!] | Ord a
ssort [!!]       = [!!]
ssort [!p:ps!]   = ssort [!x \\ x<|-ps |x<=p!] ++ [!p!] ++ ssort [!x \\ x <|-ps | x>p!]
```

Overloaded ZF-expression sorting any list (lazy, strict or unboxed).

```
gsort:: (l a) -> (l a) | Ord a & List l a
gsort [|]        = [|]
gsort [|p:ps]    = gsort [|x \\ x<|-ps | x<=p] ++ [|p] ++ gsort [|x \\ x <|-ps | x>p]
```

ZF-expression: converting an array into a list.

```
ArrayA = {1,2,3,4,5}

ListA:: [a]
ListA = [a \\ a <-: ArrayA]
```

Local definitions can be introduced in a ZF expression after generator(s) using `, let`:

Qualifier	=	Generators { `, let {` {LocalDef}+ `}` } {`|`Guard}

The variables introduced in this way can be used in the guard and local definitions, other qualifiers that follow, and the expression before the `\\`.

4.2.2 List Patterns

An object of the predefined algebraic type *list* can be specified as pattern in a function, case expression or list generator.

ListPattern	=	`[`[ListKind][{LGraphPattern}-list [`:` GraphPattern]] [SpineStrictness]`]`	
LGraphPattern	=	GraphPattern	
	|	CharsDenotation	
ListKind	=	`!`	// head strict list
	|	`#`	// unboxed head strict list
	|	`|`	// overloaded list
SpineStrictness	=	`!`	// spine strict list

Notice that only simple list patterns can be specified (one cannot use a dot-dot expression or list comprehension to define a list pattern).

Use of list patterns, use of guards, use of variables to identify patterns and sub-patterns; merge merges two (sorted) lazy lists into one (sorted) list.

```
merge:: [Int] [Int] -> [Int]
merge f []   = f
merge [] s   = s
merge f=:[x:xs] s=:[y:ys]
| x<y        = [x:merge xs s]
| x==y       = merge f ys
| otherwise  = [y:merge f ys]
```

`merge_u` merges two (sorted) head strict unboxed lists into one strict unboxed (sorted) list.

```
merge_u:: [#Int] [#Int] -> [#Int]
merge_u f [#] = f
merge_u [#] s = s
merge_u f=:[#x:xs] s=:[#y:ys]
| x<y         = [#x:merge_u xs s]
| x==y        = merge_u f ys
| otherwise   = [#y:merge_u f ys]
```

`merge_l` merges two (sorted) lists into one (sorted) list. Any list (lazy, strict or unboxed) can be merged, but both lists have to be the same (both lazy, strict, head strict, tail strict or unboxed)

```
merge_l:: (l a)(l a) -> (l a) | List l a
merge_l f [|] = f
merge_l [|] s = s
merge_l f=:[|x:xs] s=:[|y:ys]
| x<y         = [|x:merge_l xs s]
| x==y        = merge_l f ys
| otherwise   = [|y:merge_l f ys]
```

4.3 Tuples

A *tuple* is an algebraic data type predefined for reasons of programming convenience and efficiency. Tuples have as advantage that they allow bundling a *finite number* of objects of *arbitrary type* into a new object without being forced to define a new algebraic type for such a new object. This is in particular handy for functions that return several values.

4.3.1 Creating Tuples

Tuples can be created that can be used to combine different (sub-)graphs into one data structure without being forced to define a new type for this combination. The elements of a tuple need *not* be of the same type. Tuples are in particular handy for functions that return multiple results.

Tuple = (GraphExpr,{GraphExpr}-list)

Example of a Tuple of type `(String,Int,[Char])`.

```
("this is a tuple with",3,['elements'])
```

One can turn a lazy tuple element into a strict one by putting strictness annotations in the corresponding type instance on the tuple elements that one would like to make strict. When the corresponding tuple is put into a strict context the tuple and the strict annotated tuple elements will be evaluated. As usual one has to take care that these elements do not represent an infinite computation.

Strict and lazy tuples are regarded to be of different type. *However, unlike is the case with any other data structure, the compiler will automatically convert strict tuples into lazy ones, and the other way around.* This is done for programming convenience. Due to the complexity of this automatic transformation, the conversion is done for tuples only! For the programmer it means that he can freely mix tuples with strict and lazy tuple elements. The type system will not complain when a strict tuple is offered while a lazy tuple is required. The compiler will automatically insert code to convert non-strict tuple elements into a strict version and backwards whenever this is needed.

Example of a complex number as tuple type with strict components.

```
::Complex :== (!Real,!Real)

(+) infixl 6:: !Complex !Complex -> Complex
(+) (r1,i1) (r2,i2) = (r1+r2,i1+i2)
```

which is equivalent to

```
(+) infixl 6:: !(!Real,!Real) !(!Real,!Real) -> (!Real,!Real)
(+) (r1,i1) (r2,i2) = (r1+r2,i1+i2)
```

when for instance G is defined as

```
G:: Int -> (Real,Real)
```

than the following application is approved by the type system:

```
Start = G 1 + G
```

4.3.2 Tuple Patterns

An object of the predefined algebraic type *tuple* can be specified as pattern in a function, case expression or list generator.

TuplePattern	=	(GraphPattern,{GraphPattern}-list)

Example of the use of a pattern match on Tuples to access components of a Complex number.

```
:: complex :== (Real,Real)          // complex is defined as type synonym for (real,Real)

realpart:: Complex -> Real
realpart (re,_) = re                // selecting the real part of a Complex number

imagpart:: Complex -> Real
imagpart (_,im) = im                // selecting the imaginary part of a Complex number
```

4.4 Arrays

An *array* is an algebraic data type predefined for reasons of efficiency. Arrays contain a *finite number* of elements that all have to be of the *same type*. An array has as property that its elements can be accessed via *indexing* in *constant time*. An *array index* must be an integer value between `0` and the number of elements of the array-`1`. Destructive update of array elements is possible thanks to uniqueness typing. For programming convenience special syntax is provided for the creation, selection and updating of array elements (array comprehensions) while there is also special syntax for *strings* (i.e. unboxed arrays of characters). Arrays have as disadvantage that their use increases the possibility of a runtime error (indices that might get out-of-range).

To obtain optimal efficiency in time and space, arrays are implemented different depending on the concrete type of the array elements. By default an array is implemented as a *lazy array* (type `{a}`), i.e. an array consists of a contiguous block of memory containing pointers to the array elements. The same representation is chosen if *strict arrays* (define its type as `{!a}`) are being used. The difference is that *all* elements of a strict array will be evaluated if the array is evaluated. It makes no sense to make a distinction between head strictness and tail strictness for arrays as is the case for lists. As usual one has to take care that the strict elements do not represent an infinite computation.

For elements of basic type, record type and tuple type an *unboxed array* (define its type as `{#a}`) can be used. In that latter case the pointers are replaced by the array elements themselves. Lazy, strict and unboxed arrays are considered to be objects of different type. However, most predefined operations on arrays are overloaded such that they can be used on lazy, on strict as well as on unboxed arrays.

ArrayType	=	{[ArrayKind] Type}	
ArrayKind	=	!	// strict array
	|	#	// unboxed array

4.4.1 Creating Arrays and Selection of field Elements

An *array* is a tuple/record-like data structure in which *all* elements are of the *same type*. Instead of selection by position or field name the elements of an array can be selected very efficiently in *constant time* by indexing. The update of arrays is done destructively in CLEAN and therefore arrays have to be unique (see Chapter 9) if one wants to use this feature. Arrays are very useful if time and space consumption is becoming very critical (CLEAN arrays are implemented very efficiently). If efficiency is not a big issue we recommend *not* to use arrays but to use lists instead: lists induce a much better programming style. Lists are more flexible and less error prone: array elements can only be accessed via indices and if you make a calculation error, indices may point outside the array bounds. This is detected, but only at run-time. In CLEAN, array indices always start with 0. More dimensional arrays (e.g. a matrix) can be defined as an array of arrays.

For efficiency reasons, arrays are available of several types: there are *lazy arrays* (type `{a}`), *strict arrays* (type `{!a}`) and *unboxed arrays* (e.g. type`{#Int}`). All these arrays are considered to be of *different* type. By using the overloading mechanism (type constructor classes) one can still define (overloaded) functions that work on any of these arrays.

```
Array           =  ArrayDenotation
                |  ArrayUpdate
                |  ArrayComprehension
```

- All elements of an array need to be of same type.

Simple Array

A new array can be created in a number of ways. A direct way is to simply *list* the *array elements*.

```
ArrayDenotation  =  {[ArrayKind] {GraphExpr}-list}
                 |  StringDenotation

StringDenotation =  StringDel{AnyChar/StringDel}StringDel
StringDel        =  "
```

By default this array denotation is overloaded. The type determines whether a lazy, strict or unboxed array is created. The created array is *unique* (the `*` or `.` attribute in front of the type, see Chapter 9) to make destructive updates possible.

A lazy array is a box with pointers pointing to the array elements. One can also create a strict array by adding a `!` after `{` (or explicitly define its type as `{!Int}`), which will have the property that the elements to which the array box points will always be evaluated. One can furthermore create an unboxed array by adding a `#` (or explicitly define its type as `{#Int}`), which will have the property that the evaluated elements (which have to be of basic value) are stored directly in the array box itself. Clearly the last one is the most efficient representation (see also Chapter 10).

Creating a lazy array, strict and unboxed unique array of integers with elements 1,3,5,7,9.

```
MyLazyArray:: .{Int}
MyLazyArray = {1,3,5,7,9}   // overloaded array denotation

MyStrictArray:: .{!Int}
MyStrictArray = {!1,3,5,7,9}

MyUnboxedArray:: .{#Int}
MyUnboxedArray = {#1,3,5,7,9}
```

Creating a two dimensional array, in this case a unique array of unique arrays of unboxed integers.

```
MatrixA:: {.{#Int}}
MatrixA = {{1,2,3,4},{5,6,7,8}}
```

To make it possible to use operators such as array selection on any of these arrays (of actually different type) a multi parameter type constructor class has been defined (in `StdArray`) which expresses that "some kind of array structure is created". The compiler will therefore deduce the following general type:

```
Array :: .(a Int)| Array a Int
Array = {1,3,5,7,9}
```

A *string* is predefined type which equivalent to an *unboxed array of characters* `{#Char}`. Notice that this array is *not* unique, such that a destructive update of a string is *not* allowed. There is special syntax to denote strings.

Some ways to define a string, i.e. an unboxed array of character.

```
"abc"
{'a','b','c'}
```

There are a number of handy functions for the creation and manipulation of arrays predefined in CLEAN 's Standard Environment. These functions are overloaded to be able to deal with any type of array. The class restrictions for these functions express that "an array structure is required" containing "an array element".

Type of some predefined functions on `Arrays`.

```
createArray  :: !Int e ->.(a e) | Array a e            // size arg1, a.[i] = arg2
size         :: (a e) -> Int | Array a                 // number of elements in array
```

Array Update and Array Comprehensions

It is also possible to construct a new array out of an existing one (a *functional array update*).

```
ArrayUpdate          =  { ArrayExpr & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list]}
ArrayComprehension   =  {[ArrayKind] GraphExpr \\ {Qualifier}-list}
Selection            =  .FieldName
                     |  .ArrayIndex
ArrayExpr            =  GraphExpr
```

Left from the `& (a & [i] = v` is pronounced as: array `a` with for `a.[i]` the value `v`. The old array has to be specified which has to be of unique type to make destructive updating possible. On the right from the `&` those array elements are listed in which the new array differs from the old one. One can change any element of the array or any field or array element of a record or array stored in the array. The `&`-operator is strict in its arguments.

- An array expression must be of type array.
- The array expression to the left of the update operator '`&`' should yield an object of type unique array.
- An array index must be an integer value between `0` and the number of elements of the array-`1`. An index out of this range will result in a *run-time* error.
- Unboxed arrays can only be of basic type, record type or array type.
- *A unique array of any type created by an overloaded function cannot be converted to a non-unique array.*

Important: For reasons of efficiency we have defined the updates only on arrays which are of *unique* type (`*{..}`), such that the update can always be done *destructively* (!) which is semantically sound because the original unique array is known not to be used anymore.

Creating an array with the integer elements 1,3,5,7,9 using the update operator.

```
{CreateArray 5 0 & [0] = 1, [1] = 3, [2] = 5, [3] = 7, [4] = 9}
{CreateArray 5 0 & [1] = 3, [0] = 1, [3] = 7, [4] = 9, [2] = 5}
```

One can use an *array comprehension* to list these elements compactly in the same spirit as with a list comprehension (see 4.2.1).

Array comprehensions can be used in combination with the update operator. Used in combination with the update operator the original uniquely typed array is updated destructively. The combination of array comprehensions and update operator makes it possible to selectively update array elements on a high level of abstraction.

Creating an array with the integer elements 1,3,5,7,9 using the update operator in combination with array and list comprehensions.

```
{CreateArray 5 0 & [i] = 2*i+1 \\ i <- [0..4]}
{CreateArray 5 0 & [i] = elem \\ elem <-: {1,3,5,7,9} & i <- [0..4]}
{CreateArray 5 0 & elem \\ elem <-: {1,3,5,7,9}}
```

Array comprehensions used without update operator automatically generate a whole new array. The size of this new array will be equal to the size of the first array or list generator from which elements are drawn. Drawn elements that are rejected by a corresponding guard result in an undefined array element on the corresponding position.

Creating an array with the integer elements 1,3,5,7,9 using array and list comprehensions.

```
{elem \\ elem <-: {1,3,5,7,9}}
{elem \\ elem <-  [1,3,5,7,9]}
```

Array creation, selection, update). The most general types have been defined. One can of course always restrict to a more specific type.

```
MkArray:: !Int (Int -> e) ->.(a e) | Array a e
MkArray i f = {f j \\ j <- [0..i-1]}

SetArray:: *(a e) Int e ->.(a e) | Array a e
SetArray a i v = {a & [i] = v}

CA:: Int e ->.(a e) | Array a e
CA i e = createArray i e

InvPerm:: {Int} ->.{Int}
InvPerm a = {CA (size a) 0 & [a.[i]] = i \\ i <- [0..maxindex a]}

ScaleArray:: e (a e) ->.(a e) | Array a e & Arith e
ScaleArray x a = {x * e \\ e <-: a}

MapArray:: (a -> b) (ar a) ->.(ar b) | Array ar a & Array ar b
MapArray f a = {f e \\ e <-: a}

inner:: (a e) (a e) ->.(a e) | Array a e & Arith e
inner v w
| size v == size w    = {vi * wi \\ vi <-: v & wi <-: w}
| otherwise           = abort "cannot take inner product"

ToArray:: [e] ->.(a e) | Array a e
ToArray list = {e \\ e <- list}

ToList:: (a e) ->.[e] | Array a e
ToList array = [e \\ e <-: array]
```

Example of operations on 2 dimensional arrays generating new arrays.

```
maxindex n :== size n - 1

Adj:: {{#Int}} ->.{.{#Int}}
Adj ma = {{ma.[i,j] \\ i <- rowindex} \\ j <- colindex}
    where
    rowindex = [0..maxindex ma]
    colindex = [0..maxindex ma.[0]]

    Multiply:: {{#Int}} {{#Int}} ->.{.{#Int}}
    Multiply a b =   {  {sum [a.[i,j]*b.[j,k] \\ j <- colindex] \\ k <- rowindex}
                     \\ i <- rowindex
                     }
    where
        rowindex = [0..maxindex a]
        colindex = [0..maxindex a.[0]]
```

Updating unique arrays using a unique array selection.

```
MyArray:: .{#Real}
MyArray = {1.5,2.3,3.4}

ScaleArrayElem:: *{#Real} Int Real -> .{#Real}
ScaleArrayElem ar i factor
# (elem,ar) = ar![i]
= {ar & [i] = elem*factor}

Scale2DArrayElem:: {*{#Real}} (Int,Int) Real -> {.{#Real}}
Scale2DArrayElem ar (i,j) factor
# (elem,ar)       = ar![i].[j]
= {ar & [i].[j]   = elem*factor}

Scale2DArrayElem2:: {*{#Real}} (Int,Int) Real -> {.{#Real}}
Scale2DArrayElem2 ar (i,j) factor
# (elem,ar)       = ar![i,j]
= {ar & [i,j]     = elem*factor}
```

with Array Update

`variable = {variable & updates}` after `#` or `#!` can be abbreviated to `variable & updates`, by omitting `= {variable` and `}`.

```
                    |  Variable & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list] ;
```

For example

```
# a & [i] = x
```
instead of
```
# a = {a & [i] = x}
```

Multiple indices and fields are also possible, for example: (for record updates see 5.2.1 below)

```
# r & a.[i].x = y
```
instead of
```
# r = {r & a.[i].x = y}
```

Selection of an Array Element

```
ArraySelection      =  ArrayExpr . ArrayIndex {Selection}
                    |  [ArrayExpr ! ArrayIndex {Selection}
ArrayIndex          =  [{IntegerExpr}-list]
IntegerExpr         =  GraphExpr
Selection           =  .FieldName
                    |  .ArrayIndex
```

With an *array selection* (using the '`.`' symbol) one can select an array element. When an object `a` is of type `Array`, the `i`th element can be selected (computed) via `a.[i]`. Array selection is left-associative: `a.[i,j,k]` means `((a.[i]).[j]).[k]`. A "unique" selection using the '`!`' symbol returns a tuple containing the demanded array element and the original array. This type of array selection can be very handy for destructively updating of uniquely typed arrays with values that depend on the current contents of the array. Array selection binds more tightly (priority `11`) than application (priority `10`).

4.4.2 Array Patterns

An object of type *array* can be specified as pattern. Notice that only simple array patterns can be specified on the left-hand side (one cannot use array comprehensions). Only those array elements which contents one would like to use in the right-hand side need to be mentioned in the pattern.

- All array elements of an array need to be of same type.
- An array index must be an integer value between `0` and the number of elements of the array-`1`. Accessing an array with an index out of this range will result in a *run-time* error.

```
ArrayPattern    =  {{GraphPattern}-list}
                |  {{ArrayIndex = Variable}-list}
                |  StringDenotation
```

It is allowed in the pattern to use an index expression in terms of the other formal arguments (of type `Int`) passed to the function to make a flexible array access possible.

Use of array patterns.

```
Swap:: !Int !Int !*(a e) ->.(a e) | Array a e
Swap i j a=:{[i]=ai,[j]=aj} = {a & [i]=aj,[j]=ai}
```

4.5 Predefined Type Constructors

Predefined types can also be used in curried way. To make this possible a type constructor has been predefined for all predefined types of higher order kind (see also 5.1.2). The kind `X` stands for any so-called *first-order* type: a type expecting no further arguments ((`Int`, `Bool`, `[Int]`, etcetera). All function arguments are of kind `X`. The kind `X -> X` stands for a type that can be applied to a (first-order) type, which then yields another first-order type, `X -> X -> X` expects two type arguments, and so on.

```
Int, Bool, [Int], Tree [Int]   :: X
[], Tree, (,) Int, (->) a, {}  :: X -> X
(,), (->)                      :: X -> X -> X
(,,)                           :: X -> X -> X -> X
```

```
PredefinedTypeConstructor  =  []          // list type constructor
                           |  [! ]        // head strict list type constructor
                           |  [ !]        // tail strict list type constructor
                           |  [!!]        // strict list type constructor
                           |  [#]         // unboxed head strict list type
                           |  [#!]        // unboxed strict list type
                           |  ({,}+)      // tuple type constructor (arity >= 2)
                           |  {}          // lazy array type constructor
                           |  {!}         // strict array type constructor
                           |  {#}         // unboxed array type constructor
                           |  (->)        // arrow type constructor
```

So, all predefined types can be written down in prefix notation as well, as follows:

```
[] a          is equivalent with [a]
[! ] a        is equivalent with [!a]
[ !] a        is equivalent with [a!]
[!!] a        is equivalent with [!a!]
[# ] a        is equivalent with [#a]
[#!] a        is equivalent with [#a!]
(,) a b       is equivalent with (a,b)
(,,) a b c    is equivalent with (a,b,c) and so on for n-tuples
{} a          is equivalent with {a}
{!} a         is equivalent with {!a}
{#} a         is equivalent with {#a}
(->) a b      is equivalent with (a -> b)
```

4.6 Arrow Types

The *arrow type* is a predefined type constructor used to indicate *function objects* (these functions have at least arity one). One can use the Cartesian product (uncurried version) to denote the function type (see 3.7) to obtain a compact notation. Curried functions applications and types are automatically converted to their uncurried equivalent versions.

```
ArrowType  =  (Type -> Type)
```

Example of an arrow type.

```
((a b -> c) [a] [b] -> [c])
```

being equivalent with:

```
((a -> b -> c) -> [a] -> [b] -> [c])
```

4.7 Predefined Abstract And Synonym Types

Abstract data types are types of which the actual definition is hidden (see 5.4). In CLEAN the types `World` and `File` are *predefined abstract data types*. They are recognised by the compiler and treated specially, either for efficiency or because they play a special role in the language. Since the actual definition is hidden it is not possible to denotate constant values of these predefined abstract types. There are functions predefined in the CLEAN library for the creation and manipulation of these predefined abstract data types. Some functions work (only) on unique objects.

An object of type `*World` (`*` indicates that the world is unique) is automatically created when a program is started. This object is optionally given as argument to the `Start` function (see 2.3). With this object efficient interfacing with the outside world (which is indeed unique) is possible.

An object of type `File` or `*File` (unique `File`) can be created by means of the functions defined in `StdFileIO` (see CLEANs Standard Library). It makes direct manipulation of persistent data possible. The type `File` is predefined for reasons of efficiency: CLEAN `Files` are directly coupled to concrete files.

The type String is a predefined synonym type that is predefined for convenience. The type is synonym for an unboxed array of characters `{#Char}`.

```
PredefType      =  World          // see StdWorld.dcl
                |  File           // see StdFileIO.dcl
                |  String         // synonym for {#Char}
                |  ()             // unit type
```

4.8 Predefined Unit Type

The unit type is a predefined type with type constructor `()` (see 4.7) and constructor `()`.

```
UnitPattern     =  ()
UnitConstructor =  ()
```

Chapter 5

Defining New Types

CLEAN is a strongly typed language: every object (graph) and function (graph rewrite rule) in CLEAN has a type. The basic type system of CLEAN is based on the classical polymorphic Milner/Hindley/Mycroft (Milner 1978; Hindley 1969, Mycroft, 1984) type system. This type system is adapted for graph rewriting systems and extended with *basic types*, (possibly *existentially and universally quantified*) *algebraic types*, *record types*, *abstract types* and *synonym types*.

New types can be defined in an implementation as well as in a definition module. Types can *only* be defined on the global level. Abstract types can only be defined in a definition module hiding the actual implementation in the corresponding implementation module.

```
TypeDef                 =  AlgebraicTypeDef                 // see 5.1
                        |  RecordTypeDef                    // see 5.2
                        |  SynonymTypeDef                   // see 5.3
                        |  AbstractTypeDef                  // see 5.4
                        |  AbstractSynonymTypeDef
                        |  ExtensibleAlgebraicTypeDef
                        |  AlgebraicTypeDefExtension
                        |  NewTypeDef
```

5.1 Defining Algebraic Data Types

With an algebraic data type one assigns a new type constructor (a new type) to a newly introduced data structure. The data structure consists of a new constant value (called the data constructor) that can have zero or more arguments (of any type). Every data constructor must unambiguously have been (pre) defined in an algebraic data type definition Several data constructors can be introduced in one algebraic data type definition which makes it possible to define alternative data structures of the same algebraic data type. The data constructors can, just like functions, be used in a curried way. Also type constructors can be used in a curried way, albeit only in the type world of course.

Polymorphic algebraic data types can be defined by adding (possibly existentially or universally quantified, see below) type variables to the type constructors on the left-hand side of the algebraic data type definition. The arguments of the data constructor in a type definition are type instances of types (that are defined or are being defined).

Types can be preceded by uniqueness type attributes (see Chapter 9). The arguments of a defined data constructor can optionally be annotated as being strict (see 5.1.6).

```
AlgebraicTypeDef        =  ::TypeLhs  = ConstructorDef
                                      {| ConstructorDef} ;
```

```
TypeLhs                 =  [*] TypeConstructor {[*]TypeVariable}
TypeConstructor         =  TypeName
ConstructorDef          =  [ExistQuantVariables] ConstructorName {ArgType} {& ClassConstraints}
                        |  [ExistQuantVariables] (ConstructorName) [FixPrec] {ArgType} {& ClassConstraints}
FixPrec                 =  infixl [Prec]
                        |  infixr [Prec]
                        |  infix [Prec]
Prec                    =  Digit

BrackType               =  [Strict] [UnqTypeAttrib] SimpleType
ArgType                 =  BrackType
                        |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type [ClassContext])
ExistQuantVariables     =  E.{TypeVariable }+:
UnivQuantVariables      =  A.{TypeVariable }+:
```

Example of an algebraic type definition and its use.

```
::Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
::Tree a = NilTree
         | NodeTree a (Tree a) (Tree a)

MyTree:: (Tree Int)                    // constant function yielding a Tree of Int
MyTree = NodeTree 1 NilTree NilTree
```

An algebraic data type definition can be seen as the specification of a grammar in which is specified what legal data objects are of that specific type. Notice that all other CLEAN types (basic, list, tuple, array, record, abstract types) can be regarded as special cases of an algebraic type.

Constructors with two arguments can be defined as **infix** constructor, in a similar way as function operators (with fixity (**infixl**, **infixr** or just **infix**, default **infixl**) and precedence (0 through 9, default 9). If infix constructors are surrounded by brackets they can also be used in prefix position (see 3.1 and 3.4). In a pattern match they can be written down in infix position as well.

When a constructor operator is used in infix position (in an expression or in a in a pattern) *both* arguments have to be present. Constructor operators can be used in a curried way, but then they have to be used as ordinary prefix constructors (see 3.1 and 3.4).

Algebraic type definition and constructor pattern in function definition.

```
::Tree2 a    = (/\) infixl 0 (Tree a) (Tree a)
             | Value a

Mirror:: (Tree2 a) -> Tree2 a
Mirror (left/\right)  = Mirror right/\Mirror left
Mirror leaf           = leaf
```

Example of an algebraic type defining an infix data constructor and a function on this type; notice that one cannot use a '**:**' because this character is already reserved.

```
::List a = (<:>) infixr 5 a (List a)
         | Nil

Head:: (List a) -> a
Head (x<:>xs) = x
```

All data constructors being defined in the scope must have *different* names, to make type inferencing possible.

Scope of type definitions.

```
implementation module XYZ

:: Type_contructor type_vars = expression

other_definitions
```

5.1.1 Using Constructors in Patterns

An algebraic data type can be used in a pattern. The pattern consists of the *data constructor* (see 3.2) with its optional arguments which on its turn can contain *sub-patterns*. A constructor pattern forces evaluation of the corresponding actual argument to strong root normal form since the strategy has to determine whether the actual argument indeed is equal to the specified constructor.

```
GraphPattern    =  QConstructor {Pattern}          // Constructor pattern
                |  GraphPattern QConstructorName   // Infix Constructor operator
                   GraphPattern
                |  Pattern
```

Example of an algebraic data type definition and its use in a pattern in function definition.

```
::Tree a = Node a (Tree a) (Tree a)
         | Nil

Mirror:: (Tree a) -> Tree a
Mirror (Node e left right) = Node e (Mirror right) (Mirror left)
Mirror Nil                 = Nil
```

- The data constructor used in a patter must have been defined in an algebraic data type definition.

5.1.2 Using Higher Order Types

In an algebraic type definition ordinary types can be used (such as a basic type, e.g. `Int`, or a list type, e.g. `[Int]`, or an instantiation of a user defined type, e.g. Tree Int), but one can also use *higher order types*. Higher order types can be constructed by curried applications of the type constructors. Higher order types can be applied in the type world in a similar way as higher order functions in the function world. The use of higher order types increases the flexibility with which algebraic types can be defined. Predefined types can also be used in curried way (see 4.5). Higher order types play an important role in combination with type classes (see Chapter 6).

Of course, one needs to ensure that all types are applied in a correct way. To be able to specify the rules that indicate whether a type itself is correct, we introduce the notion of *kind*. A kind can be seen as the 'type of a type' In our case, the kind of a type expresses the number of type arguments this type may have. The kind `X` stands for any so-called *first-order* type: a type expecting no further arguments ((`Int`, `Bool`, `[Int]`, etcetera). All function arguments are of kind `X`. The kind `X -> X` stands for a type that can be applied to a (first-order) type, which then yields another first-order type, `X -> X -> X` expects two type arguments, and so on.

In CLEAN each *top-level* type should have kind `X`. A top-level type is a type that occurs either as an argument or result type of a function or as argument type of a data constructor (in some algebraic type definition). The rule for determining the kinds of the type variables (which can be of any order) is fairly simple: The kind of a type variable directly follows from its use. If a variable has no arguments, its kind is `X`. Otherwise its kind corresponds to the number of arguments to which the variable is applied. The kind of type variable determines its possible instantiations, i.e. it can only be instantiated with a type, which is of the same kind as the type variable itself.

Example of an algebraic type using higher order types; the type variable `t`in the definition of `Tree2` s of kind `X -> X`. `Tree2` is instantiated with a list (also of kind `X -> X`) in the definition of `MyTree2`.

```
::Tree2 t    = NilTree
             | NodeTree (t Int) (Tree2 t) (Tree2 t)

MyTree2:: Tree2 []
MyTree2 = NodeTree [1,2,3] NilTree NilTree
```

5.1.3 Defining Algebraic Data Types with Existentially Quantified Variables

An algebraic type definition can contain *existentially quantified type variable s* (or, for short, existential type variables) (**Läufer** 1992). These special variables are defined on the right-hand side of the type definition and are indicated by preceding them with "`E.`". Existential types are useful if one wants to create (recursive) data structures in which objects of *different types* are being stored (e.g. a list with elements of different types).

```
AlgebraicTypeDef     =  ::TypeLhs  = ConstructorDef
                                     {| ConstructorDef} ;
ConstructorDef       =  [ExistQuantVariables] ConstructorName {ArgType} {& ClassConstraints}
                     |  [ExistQuantVariables] (ConstructorName) [FixPrec] {ArgType} {& ClassConstraints}
ExistQuantVariables  =  E.{TypeVariable}+:
```

Example of the use of an existentially quantified type. In this example a record (see 5.2) is defined containing an existentially quantified state `s`, a method to change this state s, and a method to convert the state `s` into a `String`. Notice that upon creation of the record `MyObject` the type of the internal state and the methods defined on the state are consistent (in this case the state is of type `Int`). The methods stored in the object `Object` can (only) be applied on the state of that object thus enabling an object-oriented style of programming. The concrete type of the state hidden in the object is not visible from outside. To show it to the outside world one has to convert the state, which can be of any type, to an ordinary not existentially quantified type. For instance,`PrintState` converts the any state into a `String`. Objects that have states of different type are considered to be of the same type and can for instance be part of the same list.

```
::Object = E.s: {state::s, method::s->s, tostring:: s -> String }

MyObject =   { state = 3
             , method = (+) 1
             , tostring = toString
             }

IncrementObject obj=:{method,state} = {obj & state = method state}

PrintState obj=:{tostring,state} = tostring state

Start = PrintState (IncrementObject MyObject)        // the result will be 4
```

To ensure correctness of typing, there is a limitation imposed on the use of *existentially quantified data* structures.

Once a data structure containing existentially quantified parts is created the type of these components are forgotten. This means that, in general, if such a data structured is passed to another function it is statically impossible to determine the actual types of those components: it can be of any type. Therefore, a function having an existentially quantified data structure as input is not allowed to make specific type assumptions on the parts that correspond to the existential type variables. This implies that one can only instantiate an existential type variable with a concrete type when the object is created. In all other cases it can only be unified with a universally quantified type.

Counter Example. Illegal use of an object with existentially quantified components.

```
Start = (IncrementObject MyObject).state
```

5.1.4 Defining Algebraic Data Types with Universally Quantified Variables

An algebraic type definition can contain *universally quantified type variables* (or, for short, universal type variables) of Rank 2 (on the argument position of a data constructor). These special variables are defined on the right-hand side of a type definition on the arguments position of the data constructor being defined and have to be preceded by an "`A.`" (meaning: "for all"). It can be used to store polymorphic functions that work on arguments of 'any' type. The universal type is very useful for constructing dictionaries for overloaded functions (see Chapter 6).

```
AlgebraicTypeDef      =  ::TypeLhs  = ConstructorDef
                                    {| ConstructorDef} ;
ConstructorDef        =  [ExistQuantVariables] ConstructorName {ArgType} {& ClassConstraints}
                      |  [ExistQuantVariables] (ConstructorName) [FixPrec] {ArgType} {& ClassConstraints}
BrackType             =  [Strict] [UnqTypeAttrib] SimpleType
ArgType               =  BrackType
                      |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type)
UnivQuantVariables    =  A.{TypeVariable}+:
```

Counter Example. The following program is ill typed. Although an identity function is stored in `T2`, `T2` can contain any function that can be unified with `b -> b` (for instance `Int -> Int` will do). Therefore a type error is given for `f2` since `g` is applied to both an `Int` and a `Char`.

```
:: T2 b = C2 (b -> b)

f2:: (T2 b) -> (Int,Char)
f2 (C2 g) = (g 1, g 'a')

Id::a -> a
Id x = x

Start = f2 (C2 Ids)
```

Example of the use of a universally quantified type. In contrast with the example above it is now specified that `T` must contain a universally quantified function `b -> b`. The identity function `Id` can be stored in `T`, since its type `Id::a -> a` is actually a shorthand for `Id::A.a:a -> a`. A function from `Int -> Int` cannot be stored in `T` since this type is not unifiable with `A.a:a -> a`.

```
:: T = C (A.b: b -> b)

f:: (T b) -> (Int,Char)
f (C g) = (g 1, g 'a')

Id::a -> a
Id x = x

Start = f (C Ids)
```

5.1.5 Extensible Algebraic Types

An extensible algebraic type is an algebraic type that can be extended with additional constructors in other modules. To define it add `| ..` to an algebraic type definition (or just `..` after `|` without constructors). In other modules additional constructors may be added (once per module) by using `|` in the definition instead of `=`.

```
ExtensibleAlgebraicTypeDef  =  ::TypeLhs = {ConstructorDef |} ..;
AlgebraicTypeDefExtension   =  ::TypeLhs | ConstructorDef {| ConstructorDef} ;
```

For example, to define extensible algebraic types `X` and `Y` with constructor `A`:

```
:: X a = ..
:: Y = A Int | ..
```

To extended `X` with constructor `Xn` and `Y` with `B` and `C` in another module (in which `X` and `Y` are imported):

```
:: X a | Xn a
:: Y | B Int Int | C Int Int Int
```

Additional constructors can also be exported and imported. These constructors can be imported directly using an explicit import, but not using a ConstructorsOrFields list of an imported type

5.1.6 Strictness Annotations in Type Definitions

Functional programs will generally run much more efficient when strict data structures are being used instead of lazy ones. If the inefficiency of your program becomes problematic one can think of changing lazy data structures into strict ones. This has to be done by hand in the definition of the type.

```
AlgebraicTypeDef    =  ::TypeLhs  = ConstructorDef
                                  {| ConstructorDef} ;
ConstructorDef      =  [ExistQuantVariables] ConstructorName {ArgType} {& ClassConstraints}
                    |  [ExistQuantVariables] (ConstructorName) [FixPrec] {ArgType} {& ClassConstraints}
Strict              =  !
```

In the type definition of a constructor (in an algebraic data type definition or in a definition of a record type) the arguments of the data constructor can *optionally* be annotated as being strict. So, some arguments can be defined strict while others can be defined as being lazy. In reasoning about objects of such a type it will always be true that the annotated argument will be in strong root normal form when the object is examined. Whenever a new object is created in a strict context, the compiler will take care of the evaluation of the strict annotated arguments. When the new object is created in a lazy context, the compiler will insert code that will take care of the evaluation whenever the object is put into a strict context. If one makes a data structure strict in a certain argument, it is better not define infinite instances of such a data structure to avoid non-termination.

So, in a type definition one can define a data constructor to be strict in zero or more of its arguments. Strictness is a property of data structure that is specified in its type.

> In general (with the exceptions of tuples) one cannot arbitrary mix strict and non-strict data structures because they are considered to be of different type.

When a strict annotated argument is put in a strict context while the argument is defined in terms of another strict annotated data structure the latter is put in a strict context as well and therefore also evaluated. So, one can change the default *lazy semantics* of CLEAN into a (*hyper*) *strict semantics* as demanded. The type system will check the consistency of types and ensure that the specified strictness is maintained.

There is no explicit notation for creating unboxed versions of an algebraic data type. The compiler will automatically choose the most efficient representation for a given data type. For algebraic data type definitions containing strict elements of basic type, record type and array type an unboxed representation will be chosen.

Example: both integer values in the definition of `Point` are strict and will be stored unboxed since they are known to be of basic type. The integer values stored in `MyPoint` are strict as well, but will be stored unboxed since `MyTuple` is polymorphic.

```
::Point = (!Int,!Int)

::MyTuple a = Pair !a !a

::MyPoint :== MyTuple Int
```

A user defined lazy list similar to type `[a]` could be defined in algebraic type definition as follows:

```
::LazyList a      =   LazyCons a (LazyList a)
                  |   LazyNil
```

A head strict list similar to type `[!a]` could be defined in algebraic type definition as follows:

```
::HeadSList a     =   HeadSCons !a (HeadSList a)
                  |   HeadSNil
```

A tail strict list similar to type `[a!]` could be defined in algebraic type definition as follows:

```
::TailSList a     =   TailSCons a !(TailSList a)
                  |   TailSNil
```

A strict list similar to type `[!a!]` could be defined in algebraic type definition as follows:

```
::StrictList a    =   StrictCons !a !(StrictList a)
                  |   StrictNil
```

An unboxed list similar to type `[#Int]`could be defined in algebraic type definition as follows:

```
::UnboxedIList    =   UnboxedICons !Int  UnboxedIList
                  |   UnboxedINil
```

An unboxed list similar to type `[#Int!]` could be defined in algebraic type definition as follows:

```
::UnboxedIList    =   UnboxedICons !Int !UnboxedIList
                  |   UnboxedINil
```

5.1.7 Semantic Restrictions on Algebraic Data Types

Other semantic restrictions on algebraic data types:

- The name of a type must be different from other names in the same scope and name space (see 2.1).
- The name of a type variable must be different from other type variable names in the same scope and name space
- All type variables used on the right-hand side are bound, i.e. must either be introduced on the left-hand side of the algebraic type being defined, or they must be bound by an existential quantifier on the right-hand side, or, they must be bound by a universal quantifier specified on the corresponding argument.
- A data constructor can only be defined once within the same scope and name space. So, each data constructor unambiguously identifies its type to make type inferencing possible.
- When a data constructor is used in infix position both arguments have to be present. Data constructors can be used in a curried way in the function world, but then they have to be used as ordinary prefix constructors.
- Type constructors can be used in a curried way in the type world; to use predefined bracket-like type constructors (for lists, tuples, arrays) in a curried way they must be used in prefix notation.
- The right-hand side of an algebraic data type definition should yield a type of kind `X`, all arguments of the data constructor being defined should be of kind `X` as well.
- A type can only be instantiated with a type that is of the same kind.
- An existentially quantified type variable specified in an algebraic type can only be instantiated with a concrete type (= not a type variable) when a data structure of this type is created.

5.2 Defining Record Types

A *record type* is basically an algebraic data type in which exactly one constructor is defined. Special about records is that a *field name* is attached to each of the arguments of the data constructor and that

- records cannot be used in a curried way.

Compared with ordinary algebraic data structures the use of records gives a lot of notational convenience because the field names enable *selection by field name* instead of *selection by position*. When a record is created *all* arguments of the constructor have to be defined but one can specify the arguments in *any* order. Furthermore, when pattern matching is performed on a record, one only has to mention those fields one is interested in (see 5.2.2). A record can be created via a functional update (see 5.2.1). In that case one only has to specify the values for those fields that differ from the old record. Matching and creation of records can hence be specified in CLEAN in such a way that after a change in the structure of a record only those functions have to be changed that are explicitly referring to the changed fields.

Existential and universal type variables (see 5.1.3 and 5.1.4) are allowed in record types (as in any other type). The arguments of the constructor can optionally be annotated as being strict (see 10.1). The optional uniqueness attributes are treated in Chapter 9.

```
RecordTypeDef   =  ::TypeLhs = [ExistQuantVariables] [Strict] {{FieldName ::FieldType}-list} ;
FieldType       =  [Strict] Type
                |  UnivQuantVariables [Strict] Type
                |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type)
```

As data constructor for a record the name of the record type is used internally.

- The semantic restrictions that apply for algebraic data types also hold for record types.
- The field names inside one record all have to be different. It is allowed to use the same field name in different records. If the same names are used in different records, one can explicitly specify the intended record type when the record is constructed.

Example of a record definition.

```
::Complex    =    { re :: Real
                  , im :: Real
                  }
```

The combination of existential type variables in record types are of use for an object oriented style of programming.

Example of the use of an existentially quantified record. One can create an object of a certain type that can have different representations.

```
::Object = E.x:  { state  :: x
                 , get :: x -> Int
                 , set :: x Int -> x
                 }

CreateObject1:: Object
CreateObject1 = {state = [], get = myget, set = myset}
where
    myget:: [Int] -> Int
    myget [i:is] = i
    myget []     = 0

    myset:: [Int] Int -> [Int]
    myset is i = [i:is]
```

```
CreateObject2 = {state = 0.0, get = myget, set = myset}
where
    myget:: Real -> Int
    myget r = toInt r

    myset:: Real Int -> Real
    myset r i = r + toReal i

Get:: Object -> Int
Get {state,get} = get state

Set:: Object Int -> Object
Set o=:{state,set} i = {o & state = set state i}

Start:: [Object]
Start = map (Set 3) [CreateObject1,CreateObject2]
```

Example of a complex number as record type with strict components.

```
::Complex    =   {    re:: !Real
                 ,    im:: !Real
                 }

(+) infixl 6:: !Complex !Complex -> Complex
(+) {re=r1,im=i1} {re=r2,im=i2} = {re=r1+r2,im=i1+i2}
```

The compiler often unboxes records to make the generated code more efficient. However in some cases this is less efficient, for example for abstract data types, large unique records that can be updated in place, or when records are shared. Therefore unboxing can be prevented by adding an ! before the { in the record type definition.

5.2.1 Creating Records and Selection of Record Fields

A *record* is a tuple-like algebraic data structure that has the advantage that its elements can be selected by field name rather than by position.

```
Record          =   RecordDenotation
                |   RecordUpdate
```

Simple Records

The first way is to create a record is by *explicitly* defining a value for *each* of its fields.

```
RecordDenotation    =   {[QTypeName |] {FieldName = GraphExpr}-list]}
```

Creation of a record.

```
::Point      =   { x:: Real                          // record type definition
                 , y:: Real
                 }
::ColorPoint =   { p:: Point                         // record type definition
                 , c:: Color
                 }
::Color      =   Red | Green | Blue                  // algebraic type definition

CreateColorPoint:: (Real,Real) Color -> ColorPoint  // type of function
CreateColorPoint (px,py) col = { c = col            // function creating a new record
                               , p = { x = px
                                     , y = py
                                     }
                               }
```

- A record can only be used if its type has been defined in a record type definition; the field names used must be identical to the field names specified in the corresponding type.
- When creating a record explicitly, the order in which the record fields are instantiated is irrelevant, but *all* fields have to get a value; the type of these values must be an instantiation of the corresponding type specified in record type definition. Curried use of records is *not* possible.
- When creating a record, its type constructor that can be used to disambiguate the record from other records; the type constructor can be left out if there is *at least* one field name is specified which is not being defined in some other record.

Record Update

The second way is to construct a new record out of an existing one (a *functional record update*).

```
RecordUpdate    =  {[QTypeName|][RecordExpr &][{FieldName {Selection} = GraphExpr}-list]}
Selection       =  .FieldName
                |  .ArrayIndex
RecordExpr      =  GraphExpr
```

- The record expression must yield a record.

The record written to the left of the `& (r & f = v` is pronounced as: `r` with for `f` the value `v`) is the record to be duplicated. On the right from the `&` the structures are specified in which the new record *differs* from the old one. A structure can be any field of the record or a selection of any field or array element of a record or array stored in this record. All other fields are duplicated and created*implicitly*. Notice that the functional update is not an update in the classical, destructive sense since a *new* record is created. The functional update of records is performed very efficient such that we have not added support for destructive updates of records of unique type. The `&`-operator is strict in the record argument and arguments for strict fields.

Updating a record within a record using the functional update.

```
MoveColorPoint:: ColorPoint (Real,Real) -> ColorPoint
MoveColorPoint cp (dx,dy) = {cp & p.x = cp.p.x + dx, p.y = c.p.y + dy}
```

with Record Update

`variable = {variable & updates}` after `#` or `#!` can be abbreviated to `variable & updates`, by omitting `= {variable` and `}` (same as for array updates in section 4.4.1).

```
                |  Variable & {FieldName {Selection} = GraphExpr}-list ;
```

For example:

```
# r & x = 1
```
instead of
```
# r = {r & x = 1}
```

Multiple updates are also allowed, for example:

```
# r & x=1, y=2, z.c='a'
```
instead of
```
# r = {r & x=1, y=2, z.c='a'}
```

Selection of a Record Field

```
RecordSelection  =  RecordExpr [.QTypeName].FieldName {Selection}
                 |  RecordExpr [.QTypeName]!FieldName {Selection}
Selection        =  .FieldName
                 |  .ArrayIndex
```

With a *record selection* (using the '.' symbol) one can select the value stored in the indicated record field. A "unique" selection using the '!' symbol returns a tuple containing the demanded record field and the original record. This type of record selection can be very handy for destructively updating of uniquely typed records with values that depend on the current contents of the record. Record selection binds more tightly (priority `11`) than application (priority `10`).

Record selection.

```
GetPoint:: ColorPoint -> Point
GetPoint cp = cp.p                                  // selection of a record field

GetXPoint:: ColorPoint -> Real
GetXPoint cp = cp.p.x                               // selection of a record field

GetXPoint2:: *ColorPoint -> (Real,.ColorPoint)
GetXPoint2 cp = cp!p.x                              // selection of a record f
```

5.2.2 Record Patterns

An object of type *record* can be specified as pattern. Only those fields which contents one would like to use in the right-hand side need to be mentioned in the pattern.

RecordPattern = {[QTypeName |] {FieldName [= GraphPattern]}-list}

- The type of the record must have been defined in a record type definition.
- The field names specified in the pattern must be identical to the field names specified in the corresponding type.
- When matching a record, the type contructor which can be used to disambiguate the record from other records, can only be left out if there is *at least* one field name is specified which is not being defined in some other record.

Use of record patterns.

```
::Tree a     =    Node (RecTree a)
             |    Leaf a
::RecTree a  =    { elem  :: a
                  , left  :: Tree a
                  , right  :: Tree a
                  }

Mirror:: (Tree a) -> Tree a
Mirror (Node tree=:{left=l,right=r})    = Node {tree & left=r,right=l}
Mirror leaf                             = leaf
```

The first alternative of function `Mirror` defined in another equivalent way:

```
Mirror (Node tree=:{left,right}) = Node {tree & left=right,right=left}
```

or (except `tree` may be evaluated lazily):

```
Mirror (Node tree) = Node {tree & left=tree.right,right=tree.left}
```

5.3 Defining Synonym Types

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

SynonymTypeDef = ::TypeLhs :== Type ;

- For the left-hand side the same restrictions hold as for algebraic types.
- Cyclic definitions of synonym types (e.g. `::T a b :== G a b; ::G a b :== T a b`) are not allowed.

Example of a type synonym definition.

```
::Operator a :== a a -> a

map2:: (Operator a) [a] [a] -> [a]
map2 op [] []             = []
map2 op [f1:r1] [f2:r2]   = [op f1 f2 :map2 op r1 r2]

Start:: Int
Start = map2 (*) [2,3,4,5] [7,8,9,10]
```

5.4 Defining Abstract Data Types

A type can be exported by defining the type in a CLEAN definition module (see Chapter 2). For software engineering reasons it sometimes better only to export the name of a type but not its concrete definition (the right-hand side of the type definition). The type then becomes an *abstract data type*. In CLEAN this is done by specifying only the left-hand-side of a type in the definition module while the concrete definition (the right-hand side of the type definition) is hidden in the implementation module. So, CLEAN's module structure is used to hide the actual implementation. When one wants to do something useful with objects of abstract types one needs to export functions that can create and manipulate objects of this type as well.

- Abstract data type definitions are only allowed in definition modules, the concrete definition has to be given in the corresponding implementation module.
- The left-hand side of the concrete type should be identical to (modulo alpha conversion for variable names) the left-hand side of the abstract type definition (inclusive strictness and uniqueness type attributes).

```
AbstractTypeDef          =  :: [!][UnqOrCoercibleTypeAttrib] TypeConstructor {[*]TypeVariable};
UnqOrCoercibleTypeAttrib =  *
                         |  .
```

Example of an abstract data type.

```
definition module stack

::Stack a

Empty    ::    Stack a
isEmpty  ::   (Stack a) -> Bool
Top      ::   (Stack a) -> a
Push     :: a (Stack a) -> Stack a
Pop      ::   (Stack a) -> Stack a
```

```
implementation module stack

::Stack a :== [a]

Empty:: Stack a
Empty = []

isEmpty:: (Stack a) -> Bool
isEmpty [] = True
isEmpty s  = False

Top:: (Stack a) -> a
Top [e:s] = e

Push:: a (Stack a) -> Stack a
Push e s = [e:s]

Pop:: (Stack a) -> Stack a
Pop [e:s] = s
```

5.4.1 Defining Abstract Data Types with Synonym Type Definition

Because the concrete definition of an abstract data type does not appear in the definition module, the compiler cannot generate optimal code. Therefore, if the concrete type is a synonym type, the right-hand-side of the definition may be included surrounded by brackets:

```
AbstractSynonymTypeDef  =  AbstractTypeDef (:== Type ) ;
```

The type of the implementation is still hidden as for other abstract data types, except that the compiler uses it only to generate the same code as for a synonym type.

Chapter 6

Overloading

CLEAN allows functions and operators to be *overloaded*. *Type classes* and type constructor classes are provided (which look similar to Haskell (Hudak et al. 1992) and Gofer (Jones, 1993), although our classes have slightly different semantics) with which a restricted context can be imposed on a type variable in a type specification.

If one defines a function it should in general have a name that is *different* from all other function names defined within the same scope and name space (see 2.1). However, it is sometimes very convenient to *overload* certain functions and operators (e.g. +,−,==), i.e. use *identical* names for *different* functions or operators that perform *similar tasks* albeit on objects of *different types*.

In principle it is possible to simulate a kind of overloading by using records. One simply defines a record (see 5.2) in which a collection of functions are stored that somehow belong to each other. Now the field name of the record can be used as (overloaded) synonym for any concrete function stored on the corresponding position. The record can be regarded as a kind of *dictionary* in which the concrete function can be looked up.

Example of the use of a dictionary record to simulate overloading. *sumlist* can use the field name add as synonym for any concrete function obeying the type as specified in the record definition. The operators `+.`, `+^`, `-.` and `-^` are assumed to be predefined primitives operators for addition and subtraction on the basic types `Real` and `Int`.

```
::Arith a =  {    add      :: a a -> a
             ,    subtract :: a a -> a
             }

ArithReal = { add = (+.), subtract = (-.) }
ArithInt  = { add = (+^), subtract = (-^) }

sumlist:: (Arith a) [a] [a] -> [a]
sumlist arith [x:xs] [y:ys]    =  [arith.add x y:sumlist arith xs ys]
sumlist arith x y              =  []

Start = sumlist ArithInt [1..10] [11..20]
```

A disadvantage of such a dictionary record is that it is syntactically not so nice (e.g. one explicitly has to pass the record to the appropriate function) and that one has to pay a huge price for efficiency (due to the use of higher order functions) as well. CLEAN's overloading system as introduced below enables the CLEAN system to automatically create and add dictionaries as argument to the appropriate function definitions and function applications. To avoid efficiency loss the CLEAN compiler will substitute the intended concrete function for the overloaded function application where possible. In worst case however CLEAN's overloading system will indeed have to generate a dictionary record that is then automatically passed as additional parameter to the appropriate function.

6.1 Type Classes

In a *type class definition* one gives a name to a *set of overloaded functions* (this is similar to the definition of a type of the dictionary record as explained above). For each *overloaded function* or *operator* which is a *member* of the class the *overloaded name* and its *overloaded type* is specified. The *type class variables* are used to indicate how the different instantiations of the class vary from each other. CLEAN offers multi-parameter type constructor classes, similar to those available in Haskell.

```
TypeClassDef        =  class ClassName {[.]TypeVariable}+ [ClassContext]
                       [[where] { {ClassMemberDef}+ }] ;
                    |  class FunctionName {[.]TypeVariable}+ :: FunctionType;
                    |  class (FunctionName) [FixPrec] {[.]TypeVariable}+ :: FunctionType;
```

```
ClassMemberDef      =  FunctionTypeDef
                       [MacroDef]
```

Example of the definition of a type class; in this case the class named `Arith` contains two overloaded operators.

```
class Arith a
where
    (+) infixl 6:: a a -> a
    (-) infixl 6:: a a -> a
```

Example. Classes can have several type class variables.

```
class Arith2 a b c
where
    (:+:) infixl 6:: a b -> c
```

With an *instance declaration* an instance of a given class can be defined (this is similar to the creation of a dictionary record). When the instance is made one has to be specify for which *concrete type* an instance is created. For each overloaded function in the class an *instance of the overloaded function* or *operator* has to be defined. The type of the instance can be found via uniform substitution of the type class variables by the corresponding type instances specified in the instance definition.

```
TypeClassInstanceDef  =  instance QClassName Type+ [ClassContext]
                         [where] { {FunctionDef}+ } ;
```

Example of the definition of an instance of a type class `Arith` for type `Int`. The type of the concrete functions can be obtained via uniform substitution of the type class variable in the class definition by the corresponding type specified in the instance definition. One is not obliged to repeat the type of the concrete functions instantiated (nor the fixity or associativity in the case of operators).

```
instance Arith Int
where
    (+):: Int Int -> Int
    (+) x y = x +^ y

    (-):: Int Int -> Int
    (-) x y = x -^ y
```

Example of the definition of an instance of a type class `Arith` for type `Real`.

```
instance Arith Real
where
    (+) x y = x +. y
    (-) x y = x -. y
```

Example. Instantiation of `Arith2` using the instantiations of `Arith` specified above.

```
instance Arith2 Int  Int  Int  where (:+:) x y = x + y
instance Arith2 Int  Real Real where (:+:) x y = toReal x + y
instance Arith2 Real Int  Real where (:+:) x y = x + toReal y
instance Arith2 Real Real Real where (:+:) x y = x + y
```

One can define as many instances of a class as one likes. Instances can be added later on in any module that has imported the class one wants to instantiate.

When an instance of a class is defined a concrete definition has to be given for all the class members.

6.2 Functions Defined in Terms of Overloaded Functions

When an overloaded name is encountered in an expression, the compiler will determine which of the corresponding concrete functions/operators is meant by looking at the concrete type of the expression. This type is used to determine which concrete function to apply.

All instances of a type variable of a certain class have to be of a flat type (see the restrictions mentioned in 6.11).

If it is clear from the type of the expression which one of the concrete instantiations is meant the compiler will in principle substitute the concrete function for the overloaded one, such that no efficiency is lost.

Example of the substitution of a concrete function for an overloaded one. Given the definitions above the function

```
inc n = n + 1
```

will be internally transformed into

```
inc n = n +^ 1
```

However, it is very well possible that the compiler, given the type of the expression, cannot decide which one of the corresponding concrete functions to apply. The new function then becomes overloaded as well.

For instance, the function

```
add x y = x + y
```

becomes overloaded as well because it cannot be determined which concrete instances can be applied: `add`can be applied to arguments of any type, as long as addition (`+`) is defined on them.

This has as consequence that an additional restriction must be imposed on the type of such an expression. A *class context* has to be added to the function type to express that the function can only be applied provided that the appropriate type classes have been instantiated (in fact one specifies the type of the dictionary record which has to be passed to the function in worst case). Such a context can also be regarded as an additional restriction imposed on a type variable, introducing a kind of *bounded polymorphism*.

```
FunctionType          =  [{ArgType}+ ->] Type [ClassContext] [UnqTypeUnEqualities]
ClassContext          =  | ClassConstraints {& ClassConstraints}
ClassConstraints      =  ClassOrGenericName-list {SimpleType}+
ClassOrGenericName    =  QClassName
                      |  FunctionName { | TypeKind | }
```

Example of the use of a class context to impose a restriction on the instantiation of a type variable. The function `add` can be applied on arguments of any type under the condition that an instance of the class `Arith` is defined on them.

```
add:: a a -> a | Arith a
add x y = x + y
```

CLEAN's type system can infer class contexts automatically. If a type class is specified as a restricted context the type system will check the correctness of the specification (as always a type specification can be more restrictive than is deduced by the compiler).

6.3 Type Classes Defined in Terms of Overloaded Functions

The concrete functions defined in a class instance definition can also be defined in terms of (other) overloaded functions. This is reflected in the type of the instantiated functions. Both the concrete type and the context restriction have to be specified.

Example of an instance declaration with a type which is depending on the same type class. The function + on lists can be defined in terms of the overloaded operator +on the list elements. With this definition +is defined not only on lists, but also on a list of lists etcetera.

```
instance Arith [a] | Arith a                        // on lists
where
    (+) infixl 6:: [a] [a] -> [a] | Arith a
    (+) [x:xs] [y:ys] = [x + y:xs + ys]
    (+) _       _     = []

    (-) infixl 6:: [a] [a] -> [a] | Arith a
    (-) [x:xs] [y:ys] = [x - y:xs - ys]
    (-) _      _      = []
```

Equality class.

```
class Eq a
where
  (==) infix 2:: a a -> Bool

instance Eq [a] | Eq a                              // on lists
where
    (==) infix 2:: [a] [a] -> Bool | Eq a
    (==) [x:xs] [y:ys] = x == y && xs == ys
    (==) []     []     = True
    (==) _      _      = False
```

6.4 Type Constructor Classes

The CLEAN type system offers the possibility to use higher order types (see 3.7.1). This makes it possible to define *type constructor classes* (similar to constructor classes as introduced in Gofer, Jones (1993)). In that case the overloaded type variable of the type class is not of kind `X`, but of higher order, e.g. `X -> X`, `X -> X -> X`, etcetera. This offers the possibility to define overloaded functions that can be instantiated with type constructors of higher order (as usual, the overloaded type variable and a concrete instantiation of this type variable need to be of the same kind). This makes it possible to overload more complex functions like map and the like.

Example of a definition of a type constructor class. The class `Functor` including the overloaded function `map` which varies in type variable `f` of kind `X -> X`.

```
class Functor f
where
    map:: (a -> b) (f a) -> (f b)
```

Example of an instantiation of a type constructor class. An instantiation of the well-known function `map` applied on lists (`[]` is of kind `X -> X`), and a map function defined on `Tree`'s (`Tree` is of kind `X -> X`).

```
instance Functor []
where
    map:: (a -> b) [a] -> [b]
    map f [x:xs] = [f x : map f xs]
    map f []     = []

::Tree a = (/\) infixl 0 (Tree a) (Tree a)
         | Leaf a

instance Functor Tree
where
    map:: (a -> b) (Tree a) -> (Tree b)
    map f (l/\r)      = map f l /\ map f r
    map f (Leaf a)    = Leaf (f a)
```

CLEAN 2.0 offers the possibility to define generic functions. With generic functions one is able to define a function like `map` once that works for *any* type (see 7.2).

6.5 Overlapping Instances

Identical instances of the same class are not allowed. The compiler would not know which instance to choose. However, it is not required that all instances are of different type. It is allowed to specify an instance of a class of which the types overlap with some other instance given for that class, i.e. the types of the different class instances are different but they can be unified with each other. It is even allowed to specify an instance that works for any type, just by instantiating with a type variable instead of instantiating with a concrete type. This can be handy to define a simple default case (see also the section one generic definitions). If more than one instance is applicable, the compiler will always choose the most specific instantiation.

Example of overlapping instances. Below there are three instances given for the class `Eq`: one for `Int`eger values, one for `Real` values, and one for objects of any type. The latter instance is more general and overlaps with both the other instances that are more specific. If `Int`egers or `Real`s are compared, the corresponding equality function will be chosen. For all other types for which no specific instances for equality are defined, the general instance will be chosen.

```
class Eq a
where
    (==) infix 2:: a a -> Bool

instance Eq Int                                  // on Integers
where
    (==) x y = x ==^ y

instance Eq Real                                 // on Reals
where
    (==) x y = x ==. y

instance Eq a                                   // generic instance for Eq
where
    (==) x y = False
```

It is sometimes unclear which of the class instances is the most specific. In that case the lexicographic order is chosen looking at the specified instances (with type variables always > type constructors).

Example of overlapping instances. The two instances of class `C` overlap with each other. In the `Start` rule the function `f` is applied to two `Bool`ean values. In this case any of the two instances of `f` could be chosen. They both can be applied (one has type `f::Bool a -> Bool`, the other `f::a Bool -> Bool`, `Start` requires `f:: Bool Bool -> Bool`). The compiler will choose the first instance, because in lexicographical order `instance C Bool dontcare` <= `instance C dontcare Bool`.

```
class C a1 a2
where
    f:: a1 a2 -> Bool

instance C Bool dontcare
where
    f b1 b2 = b1

instance C dontcare Bool
where
    f b1 b2 = b2

Start = f True False                        // the result will yield True
```

6.6 Internal Overloading

It is possible that a CLEAN expression using overloaded functions is internally ambiguously overloaded. The problem can occur when an overloaded function is used which has on overloaded type in which an overloaded type variable appears on the right-hand side of the `->`. If such a function is applied in such a way that the overloaded type does not appear in the resulting type of the application, any of the available instances of the overloaded function can be used.

In that case that an overloaded function is internally ambiguously overloaded the compiler cannot determine which instance to take: a type error is given.

Counter example (ambiguous internal overloaded expression). The function body of `f` is internally ambiguously overloaded which results in a type error. It is not possible to determine whether its argument should be converted to an `Int` or to a `Bool`.

```
class Read  a:: a -> String

class Write a:: String -> a

instance Read  Int, Bool                        // export of class instance, see 6.10

instance Write Int, Bool

f:: String -> String
f x = Write (Read x)                            // ! This results in a type error !
```

One can solve such an ambiguity by splitting up the expression in parts that are typed explicitly such that it becomes clear which of the instances should be used.

```
f:: String -> String
f x = Write (MyRead x)
where
    MyRead:: Int -> String
    MyRead x = Read x
```

Counter example (ambiguous internal overloaded expression). The function `:+:` is internally ambiguously overloaded which results in a type error. The compiler is not able to infer the result type `c` of the multi parameter type class `Arith2 a b c`. The reason is that the compiler will first do the type unification and then tries to solve the overloading. In this case solving the overloading will have consequences for other overloading situations. The system can only solve one overloaded situation at a time and solving the overloading may not have any effect on other unifications.

```
Start :: Int
Start = 2 :+: 3 :+: 4
```

Example (ambiguous internal overloaded expression). By explicitly specifying types the overloading can be solved. The following program is accepted.

```
Start:: Int
Start = 2 :+: more
where
    more:: Int
    more = 3 :+: 4
```

6.7 Defining Derived Members in a Class

The members of a class consist of a set of functions or operators that logically belong to each other. It is often the case that the effect of some members (*derived members*) can be expressed in others. For instance, `<>` can be regarded as synonym for `not (==)`. For software engineering (the fixed relation is made explicit) and efficiency (one does not need to include such derived members in the dictionary record) it is good to make this relation explicit. In CLEAN the existing macro facilities (see Chapter 10.3) are used for this purpose.

Classes with macro definitions to specify derived members.

```
class Eq a
where
    (==) infix 2:: a a -> Bool

    (<>) infix 2:: a a ->  Bool | Eq a
    (<>) x y :== not (x == y)

class Ord a
where
    (<) infix 2:: a a ->  Bool

    (>) infix 2:: a a ->  Bool | Ord a
    (>) x y :== y < x

    (<=) infix 2:: a a ->  Bool | Ord a
    (<=) x y :== not (y<x)

    (>=) infix 2:: a a ->  Bool | Ord a
    (>=) x y :== not (x<y)

min:: a a -> a | Ord a
min x y :== if (x<y) x y

max:: a a -> a | Ord a
max x y :== if (x<y) y x
```

6.8 A Shorthand for Defining Overloaded Functions

A class definition seems sometimes a bit overdone when a class actually only consists of one member. Special syntax is provided for this case.

```
TypeClassDef        =  class ClassName {[.]TypeVariable}+ [ClassContext]
                       [[where] { {ClassMemberDef}+ }] ;
                    |  class FunctionName {[.]TypeVariable}+ :: FunctionType;
                    |  class (FunctionName) [FixPrec] {[.]TypeVariable}+ :: FunctionType;
```

Example of an overloaded function/operator.

```
class (+) infixl 6 a:: a a -> a
```

which is shorthand for:

```
class + a
where
     (+) infixl 6:: a a -> a
```

The instantiation of such a simple one-member class is done in a similar way as with ordinary classes, using the name of the overloaded function as class name.

Example of an instantiation of an overloaded function/operator.

```
instance + Int
where
    (+) x y = x +^ y
```

6.9 Classes Defined in Terms of Other Classes

In the definition of a class one can optionally specify that other classes that already have been defined elsewhere are included. The classes to include are specified as context after the overloaded type variable. It is not needed (but it is allowed) to define new members in the class body of the new class. In this way one can give a new name to a collection of existing classes creating a hierarchy of classes (cyclic dependencies are forbidden). Since one and the same class can be included in several other classes, one can combine classes in different kinds of meaningful ways. For an example have a closer look at the CLEAN standard library (see e.g. `StdOverloaded` and `StdClass`)

Example of defining classes in terms of existing classes. The class `Arith` consists of the class `+ and -`.

```
class (+) infixl 6 a:: a a -> a

class (-) infixl 6 a:: a a -> a

class Arith a | +,- a
```

6.10 Exporting Type Classes

To export a class one simply repeats the class definition in the definition module (see Chapter 2). To export an instantiation of a class one simply repeats the instance definition in the definition module, however *without* revealing the concrete implementation (which can only be specified in the implementation module).

```
TypeClassInstanceExportDef  =  instance ClassName InstanceExportTypes ;
InstanceExportTypes         =  {Type+ [ClassContext]}-list
                            |  Type+ [ClassContext] [where] {{FunctionTypeDef}+ }
                            |  Type+ [ClassContext] [Special]
Special                     =  special {{TypeVariable = Type}-list { ; {TypeVariable = Type}-list }}
```

Exporting classes and instances.

```
definition module example

class Eq a                          // the class Eq is exported
where
    (==) infix 2:: a a -> Bool

instance Eq [a] | Eq a              // an instance of Eq on lists is exported
special  a = Int                    // with an additional specialised version for [Int]
         a = Real                   // and an additional specialised version for [Real]

instance Eq  a                      // a general instance of Eq is exported
```

For reasons of efficiency the compiler will always make specialized efficient versions of overloaded functions inside an implementation module. For each concrete application of an overloaded function a specialized version is made for the concrete type the overloaded function is applied to. So, when an overloaded function is used in the implementation module in which the overloaded function is defined, no overhead is introduced.

However, when an overloaded function is exported it is unknown with which concrete instances the function will be applied. So, a dictionary record is constructed in which the concrete function can be stored as is explained in the introduction of this section. This approach can be very inefficient, especially in comparison to a specialized version. One can therefore ask the compiler to generate specialized versions of an overloaded function that is being exported. This can be done by using the keyword **`special`**. If the exported overloaded function will be used very frequently, we advise to specialize the function for the most important types it will be applied on.

A specialised function can only be generated if a concrete type is definend for all type variables which appear in the instance definition of a class.

The types of class instance members may be specified. Strictness annotations specified in the type of the instance member of the class definition should be included, because the annotations are copied when the type of the class instance member is instantiated.

Additional strictness annotations are permitted. For example:

```
class next a where
    next :: a -> a

instance next Int where
    next :: !Int -> Int
    next x = x+1
```

If such an instance is exported, the type of the instance member must be included in the definition module:

```
instance next Int where
    next :: !Int -> Int
```

If no additional strictness annotations are specified, it can still be exported without the type.

6.11 Semantic Restrictions on Type Classes

Semantic restrictions:

- When a class is instantiated a concrete definition must be given for each of the members in the class (not for derived members).
- The type of a concrete function or operator must exactly match the overloaded type after uniform substitution of the overloaded type variable by the concrete type as specified in the corresponding type instance declaration.
- The overloaded type variable and the concrete type must be of the same kind.
- A type instance of an overloaded type must be a *flat type*, i.e. a type of the form `T a1 ... an` where `ai` are type variables which are *all* different.
- It is not allowed to use a type synonym as instance.
- The `Start` function cannot have an overloaded type.
- For the specification of derived members in a class the same restrictions hold as for defining macros.
- A restricted context can only be imposed on one of the type variables appearing in the type of the expression.
- The specification of the concrete functions can only be given in implementation modules.
- A specialised function can only be generated if a concrete type is defined for all type variables which appear in the instance definition of a class .
- A request to generate a specialised function for a class instance can only be defined in a definition module.

Chapter 7

Generic Programming

7.1 Basic Ideas Behind Generic Programming

In the previous Chapter on overloading it is explained how type classes can be used to define *different* functions or operators that have the *same* name and perform *similar tasks* albeit on objects of *different types*. These tasks are supposed to be similar, but they are in general not exactly the same. The corresponding function bodies are often slightly different because the data structures on which the functions work differ. As a consequence, one has to explicitly specify an implementation for every concrete instance of an overloaded function.

Equality class. The equality function on lists and trees. The programmer has to specify explicitly the function bodies for each concrete instantiation of equality.

```
:: List a    =   Nil a
             |   Cons a (List a)
:: Tree a    =   Leaf a
             |   Node (Tree a) (Tree a)

class Eq a
where
   (==) infix 2 :: a a -> Bool

instance Eq (List a) | Eq a
where
   (==) infix 2 :: (List a) (List a) -> Bool | Eq a
   (==) Nil Nil                 = True
   (==) (Cons x xs) (Cons x xs)= x == y && xs == ys
   (==) _      _                = False

instance Tree a | Eq a
where
   (==) infix 2 :: (Tree a) (Tree a) -> Bool | Eq a
   (==) (Leaf x) (Leaf y)          = x == y
   (==) (Node lx rx)  (Node ly ry) = lx == ly && rx == ry
   (==) _      _                   = False
```

In the example above the programmer explicitly defines equality on lists and trees. For each new data type that we want to compare for equality, we have to define a similar instance. Moreover, if such a data type changes, the corresponding instance should be changed accordingly as well. Though the instances are *similar*, they are not the same since they operate on different data types.

What is similar in these instances? Both instances perform pattern match on the arguments. If the constructors are the same, they compare constructor arguments pairwise. For constructors with no arguments `True` is returned. For constructors with several arguments, the results on the arguments are combined with &&. If constructors are not the same, `False` is returned. In other words, equality on a data type is defined by looking at the structure of the data type. More precisely, it is defined by induction on the structure of types. There are many more functions than just equality that expose the same kind of similarity in their instances. Below is the mapping function that can be defined for type constructors of kind *->*.

The Functor provides the mapping function for a type constructor of kind *->* (see also 6.4).

```
class Functor f
where
    fmap :: (a -> b) (f a) -> (f b)

instance Functor List
where
    fmap :: (a -> b) (List a) -> (List b)
    fmap f Nil         = Nil
    fmap f (Cons x xs) = Cons (f x) : Cons (fmap f xs)

instance Functor Tree
where
    fmap :: (a -> b) (Tree a) -> (Tree b)
    fmap f (Leaf a)   = Leaf (f a)
    fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

Again, both instances are similar: they perform pattern match on constructors and pairwise mapping of their arguments. The results are packed back in the same constructor.

Generic programming enables the programmer to capture this kind of similarities and define a single implementation for *all* instances of such a class of functions. To do so we need a universal structural representation of *all* data types. A generic function can then be defined *ones and forall* on that universal representation. A specific type is handled using its structural representation. In the rest of this section we will explain roughly how it all works. First we focus on the universal structural representation of types; then we show how a generic function can be defined on it; and at last we show how the generic definition is specialized to a concrete type. See also (Alimarine & Plasmeijer, 2001, A Generic Programming Extension for Clean).

In CLEAN data types are algebraic: they are built in of sums of products of type terms. For example, the `List` type is a sum of two things: nullary product for `Nil` and a binary product of the head and the tail for `Cons`. The Tree type is a sum of unary product of elements for Leaf and binary product of trees for `Node`. Having this in mind we can uniformly represent CLEAN data types using binary sums and products.

Binary sum and product types defined in StdGeneric.dcl. These types are needed to represent CLEAN types as sums of products of types for the purpose of generic programming.

```
:: UNIT a        = UNIT a
:: PAIR a b      = PAIR a b
:: EITHER l r    = LEFT l | RIGHT r
```

The `UNIT` type represents a nullary product. The `PAIR` type is a binary product. The `EITHER` type is a binary sum. We do not need a type for nullary sums, as in CLEAN data types have at least one alternative. As one can imagine, we want sum-product representation of types to be equivalent (i.e. isomorphic) to the original data types. In the following example we give representations for `List` and `Tree` with conversion functions that implement the required isomorphisms.

Sum-product representation of the structure of `List` and `Tree` with conversion functions that implement isomorphisms between the types and their sum-product representations.

```
:: ListS a :== EITHER UNIT (PAIR a (List a))

listToStruct    :: (List a)  -> _ListS a
listToStruct Nil                  = LEFT UNIT
listToStruct (Cons x xs)          = RIGHT (PAIR x xs)

listFromStruct   :: (ListS a) -> _List a
listFromStruct (LEFT UNIT)        = Nil
listFromStruct (RIGHT (PAIR x xs)  = Cons x xs

:: TreeS a :== EITHER a (PAIR (Tree a) (Tree a))

treeToStruct     :: (Tree a)   -> _TreeS a
treeFromStruct   :: (TreeS a)  -> _Tree a
```

As we said, all algebraic types can be represented in this way. Basic types are not algebraic, but there are only few of them: they are represented by themselves. Arrow types are represented by the arrow type constructor (->). To define a function generically on the structure of a type it is enough to define instances on the components the structure can be built from. These are binary sums, binary products, basic types, and the arrow type.

Equality on sums, products and primitive types. Equality cannot be feasibly defined for the arrow type, so the instance is omitted.

```
instance UNIT
where
    (==) infix 2 :: UNIT UNIT -> Bool
    (==) UNIT UNIT            = True

instance PAIR a b | Eq a & Eq b
where
    (==) infix 2 :: (PAIR a b) (PAIR a b) -> Bool
    (==) (PAIR a1 b1) (PAIR a2 b2) = a1 == a2 && b1 == b2

instance EITHER a b | Eq a & Eq b
where
    (==) infix 2 :: (EITHER a b) (EITHER a b) -> Bool
    (==) (LEFT x)  (Leaf y)    = x == y
    (==) (RIGHT x) (RIGHT y)   = x == y
    (==) x_         y_         = False

instance Int
where
    (==) infix 2 :: Int Int -> Bool
    (==) x y = eqInt x y             // primitive equality on integers
```

Having defined instances on the structure components we can generate instances for all other types automatically.

Equality for lists and trees that can be automatically generated. Arguments are first converted to the structural representations which are then compared.

```
instance Eq (List a) | Eq a
where
    (==) infix 2 :: (List a) (List a) -> Bool | Eq a
    (==) xs ys = listToStruct xs == listToStruct ys

instance Tree a | Eq a
where
    (==) infix 2 :: (Tree a) (Tree a) -> Bool | Eq a
    (==) xs ys = treeToStruct xs == treeToStruct ys
```

Not only instances of one class expose similarity in the definition of instances.

The Bifunctor provides the mapping function for a type constructor of kind *->*->*. Instances are defined in a similar way to instances of Functor.

```
:: Tree2 a b      = Tip a
                  | Bin b (Tree a b) (Tree a b)

class Bifunctor f
where
    bmap :: (a1 -> b1) (a2 -> b2) (f a1 a2) -> (f b1 b2)

instance Bifunctor Tree
where
    bmap :: (a1 -> b1) (a2 -> b2) (Tree2 a1 a2) -> (Tree b1 b2)
    bmap f1 f2 (Tip x) = Tip (f1 x)
    bmap f1 f2 (Bin x l r) = Bin (f2 x) (bmap f1 f2 l) (bmap f1 f2 r)
```

The instance in the example above also works in a similar way as the instance of Functor: it also maps substructures component-wise. Both Functor and Bifunctor provide *mapping* function. The difference is that one provides mapping for type constructors of kind *->* and the other for type constructors of kind *->*->*. In fact instances of mapping functions for all kinds are similar.

7.2 Defining Generic Functions

The generic feature of CLEAN is able to derive instances for types of different kinds from a single generic definition. Such generic functions are known as kind-indexed generic functions (Alimarine & Plasmeijer, A Generic Programming Extension for Clean). Actually, a generic function in CLEAN stands for a *set* of classes and instances of the *same* function for *different* kinds. Since CLEAN allows function to be used in a Curried manner (see 3.7.1), the compiler is in general not able to deduce which kind of map is meant. Therefore the *kind* of a generic function application has to be specified explicitly.

To define a generic function the programmer has to provide to things: the base type of the generic function and the base cases (instances) of the generic function.

```
GenericsDef          =  GenericDef ;
                     |  GenericCase;
                     |  DeriveDef ;
```

```
GenericDef           =  generic FunctionName TypeVariable+ [GenericDependencies] :: FunctionType
GenericDependencies  =  | {FunctionName TypeVariable+ }-list
GenericCase          =  FunctionName {|GenericTypeArg|} {Pattern}+ = FunctionBody
GenericTypeArg       =  GenericMarkerType [of Pattern]
                     |  TypeName
                     |  TypeVariable
GenericMarkerType    =  CONS
                     |  OBJECT
                     |  RECORD
                     |  FIELD
```

In the generic definition, recognised by the keyword **generic**, first the type of the generic function has to be specified. The type variables mentioned after the generic function name are called *generic type variables*. Similar to type classes, they are substituted by the actual instance type. A generic definition actually defines a *set* of type constructor classes. There is one class for each possible kind in the set. Such a generic function is sometimes called a kind-indexed class. The classes are generated using the type of the generic function. The classes always have one class variable, even if the generic function has several generic variables. The reason for this restriction is that the generic function can be defined by induction on one argument only.

Example. The generic definition of equality.

```
generic gEq a ::          a             a             -> Bool
gEq {|Int|}               x             y             = x == y
gEq {|Char|}              x             y             = x == y
gEq {|Bool|}              x             y             = x == y
gEq {|Real|}              x             y             = x == y
gEq {|UNIT|}              UNIT          UNIT          = True
gEq {|PAIR|}    eqx eqy   (PAIR x1 y1)  (PAIR x2 y2)  = eqx x1 x2 && eqy y1 y2
gEq {|EITHER|} eql eqr    (LEFT x)      (LEFT y)      = eql x y
gEq {|EITHER|} eql eqr    (RIGHT x)     (RIGHT y)     = eqr x y
gEq {|EITHER|} eql eqr    x             y             = False
gEq {|CONS|}    eq        (CONS x)      (CONS y)      = eq x y
gEq {|OBJECT|} eq         (OBJECT x)    (OBJECT y)    = eq x y
gEq {|RECORD|} eq         (RECORD x)    (RECORD y)    = eq x y
gEq {|FIELD|}   eq        (FIELD x)     (FIELD y)     = eq x y
```

Example. The generic definition of map.

```
generic gMap a b ::        a             -> b
gMap {|c|}         x                     = x
gMap {|PAIR|}      fx fy    (PAIR x y)   = PAIR (fx x) (fy y)
gMap {|EITHER|}    fl fr    (LEFT x)     = LEFT (fl x)
gMap {|EITHER|}    fl fr    (RIGHT x)    = RIGHT (fr x)
gMap {|CONS|}      fx       (CONS x)     = CONS (fx x)
gMap {|OBJECT|}    fx       (OBJECT x)   = OBJECT (fx x)
gMap {|RECORD|}    fx       (RECORD x)   = RECORD (fx x)
gMap {|FIELD|}     fx       (FIELD x)    = FIELD (fx x)
```

Classes that are automatically generated for the generic map function given above.

```
class gMap{|*|} t                :: t -> t
class gMap{|*->*|} t             :: (a -> b) (t a) -> t b
class gMap{|*->*->*|} t          :: (a1 -> b1) (a2 -> b2) (t a1 a2) -> t b1 b2
...
```

Roughly the algorithm for deriving classes is the following.

Algorithm for generating classes. Suppose we have a generic function `genFun` with type `GenFun`.

```
:: GenFun a1 .. an :== ..
generic genFun a1 .. an :: GenFun a1 .. an
```

A class for kind *k*.

```
class genFun{|k|} t      :: GenFun{|k|} t .. t
```

Is derived by induction on the structure of kind

```
:: GenFun{|*|} a1 ... an   :== GenFun a1.. an
:: GenFun{|k->l|} a1 ..an  :==
    A.b1 .. bn: (GenFun{|k|} b1 .. bn) -> GenFun{|l|} (a1 b1) .. (an bn)
```

The programmer provides a set of basic cases for a generic function. Based on its basic cases a generic function can be derived for other types. See the next section for detailed discussion on types for which a generic function can and cannot be derived. Here we discuss what can be specified as the type argument in the definition of a generic base case

- *A Generic structural representation type*: `UNIT`, `PAIR`, `EITHER`, `CONS`, `OBJECT`, `RECORD` and `FIELD`. The programmer *must always provide* these cases as they cannot be derived by the compiler. Without these cases a generic function cannot be derived for basically any type.
- *Basic type*. If a generic function is supposed to work on types that involve basic types, instances for basic types must be provided.
- *Type variable*. Type variable stands for all types of kind *. If a generic function has a case for a type variable it means that by default all types of kind star will be handled by that instance. The programmer can override the default behavior by defining an instance on a specific type.
- *Arrow type* `(->)`. If a generic function is supposed to work with types that involve the arrow type, an instance on the arrow type has to be provided.
- *Type constructor*. A programmer may provide instances on other types. This may be needed for two reasons:
  1. The instance cannot be derived for the reasons explained in the next section.
  2. The instance can be generated, but the programmer is not satisfied with generic behavior for this type and wants to provide a specific behavior.

7.3 Deriving Generic Functions

The user has to tell the compiler which instances of generic functions on which types are to be generated. This is done with the *derive* clause.

DeriveDef	=	**derive** `FunctionName` {DerivableType}-list
	|	**derive class** `ClassName` {DerivableType}-list
DerivableType	=	`TypeName`
	|	PredefinedTypeConstructor

Deriving instances of generic mapping and generic equality for List , Tree and standard list

```
derive gEq  List, Tree, []
derive gMap List, Tree, []
```

A generic function can be automatically specialized only to algebraic types that are not abstract in the module where the *derive* directive is given. A generic function cannot be automatically derived for the following types:

- *Generic structure representation types:* `UNIT`, `PAIR`, `EITHER`, `CONS`, `OBJECT`, `RECORD`, `FIELD`. See also the previous section. It is impossible to derive instances for these types automatically because they are themselves used to build structural representation of types that is needed to derive an instance. Deriving instances for then would yield non-terminating cyclic functions. Instances on these types must be provided for the user. Derived instances of algebraic types call these instances.
- *Arrow type* `(->)`. An instance on the arrow type has to be provided by the programmer, if he or she wants the generic function to work with types containing arrows.
- *Basic types* like `Int`, `Char`, `Real`, `Bool`. In principle it is possible to represent all these basic types as algebraic types but that would be very inefficient. The user can provide a user defined instance on a basic type.
- *Array types* as they are not algebraic. The user can provide a user defined instance on an array type.
- *Synonym types*. The user may instead derive a generic function for the types involved on the right-hand-side of a type synonym type definition.
- *Abstract types*. The compiler does not know the structure of an abstract data type needed to derive the instance.
- *Quantified types*. The programmer has to manually provide instances for type definitions with universal and existential quantification.

The compiler issues an error if there is no required instance for a type available. Required instances are determined by the overloading mechanism.

`derive class` of a class and type derives for the type all generic functions that occur directly in the context of the class definition (see 6.1).

For example:

```
import GenEq,GenLexOrd

class C a | gEq{|*|} a & gLexOrd{|*|} a & PlusMin a

:: T = V Char

derive class C T    // derives gEq and gLexOrd for type T
```

7.4 Applying Generic Functions

The generic function in Clean stands for a set of overloaded functions. There is one function in the set for each kind. When a generic function is applied, the compiler must select one overloaded function in the set. The compiler cannot derive the required kind automatically. For this reason a kind has to be provided explicitly at each generic function application. Between the brackets `{|` and `|}` one can specify the intended kind. The compiler then resolves overloading of the selected overloaded function as usually.

```
GenericAppExpression  =  FunctionName {|TypeKind|} GraphExpr
TypeKind              =  *
                      |  TypeKind -> TypeKind
                      |  IntDenotation
                      |  (TypeKind)
                      |  {|TypeKind|}
```

Example: a generic equality operator can be defined as equality on types of kind *.

```
(===) infix 2 :: a a -> Bool | gEq{|*|} a
(===) x y = gEq{|*|} x y
```

Example: a mapping function `fmap` for functors and `bmap` for bifunctors can be defined in terms of the generic mapping function defined above just by specializing it for the appropriate kind.

```
fmap :: (a -> b) (f a) -> (f b) | gMap{|*->*|} f
fmap f x y = gMap{|*->*|} f x y

bmap :: (a1 -> b1) (a2 -> b2) (f a1 a2) -> (f b1 b2) | gMap{|*->*->*|} f
bmap f1 f2 x y = gMap{|*->*->*|} f1 f2 x y
```

Equality makes sense not only on for kind *. In the example we alter the standard way of comparing elements. Equality for kind * and *->* are explicitly used.

```
eqListFsts :: [(a, b)] [(a, c)] -> Bool | gEq{|*|} a
eqListFsts xs ys = gEq{|*->*|} (\x y -> fst x === fst y) ys

eqFsts :: (f (a, b)) (f (a, c)) -> Bool | gEq{|*->*|} f & gEq{|*|} a
eqFsts xs ys     = gEq{|*->*|} (\x y -> fst x === fst y) ys
```

Examples of generic applications

```
Start =
    ( gEq{|*|} [1,2,3] [2,3,3]                          // True
    , [1,2,3] === [1,2,3]                               // True
    , gEq{|*->*|} (\x y -> True) [1,2,3] [4,5,6]        // True
    )
```

7.5 Using Constructor Information

As it was outlined above, the structural representation of types lacks information about specific constructors and record fields, such as name, arity etc. This is because this information is not really part of the structure of types: different types can have the same structure. However, some generic functions need this information. Consider, for example a generic toString function that converts a value of any type to a string. It needs to print constructor names. For that reason the structural representation of types is extended with special constructor and field markers that enable us to pass information about fields and constructors to a generic function.

Definition of the constructor and field marker types (in StdGeneric.dcl).

```
:: CONS a         = CONS a
:: OBJECT a       = OBJECT a
:: RECORD a       = RECORD a
:: FIELD a        = FIELD a
```

The markers themselves do not contain any information about constructors and fields. Instead, the information is passed to instances of a generic function on these markers.

Examples of structural representation with constructor and field information

```
:: ListS a    :== OBJECT (EITHER (CONS UNIT) (CONS (PAIR a (List a))))
:: TreeS a    :== OBJECT (EITHER (CONS a) (CONS (PAIR (Tree a) (Tree a))))

:: Complex    = { re    :: Real, im    :: Real }
:: ComplexS   :== RECORD (PAIR (FIELD Real) (FIELD Real))
```

```
GenericTypeArg          =  GenericMarkerType [of Pattern]
                        |  TypeName
                        |  TypeVariable
GenericMarkerType       =  CONS
                        |  OBJECT
                        |  RECORD
                        |  FIELD
```

Definition of the algebraic type, constructor, record type and field descriptors (StdGeneric.dcl)
The algebraic type descriptor is passed after **of** in the OBJECT case of a generic function.

```
:: GenericTypeDefDescriptor =
    { gtd_name        :: String
    , gtd_arity       :: Int
    , gtd_num_conses  :: Int
    , gtd_conses      :: [GenericConsDescriptor]
    }
```

The constructor descriptor is passed after **of** in the CONS case of a generic function.

```
:: GenericConsDescriptor =
    { gcd_name        :: String
    , gcd_arity       :: Int
    , gcd_prio        :: GenConsPrio              // priority and associativity
    , gcd_type_def    :: GenericTypeDefDescriptor // type def of the constructor
    , gcd_type        :: GenType                  // type of the constructor
    , gcd_index       :: Int                      // index of the contructor in the type def
    }
```

The record descriptor is passed after **of** in the RECORD case of a generic function.

```
:: GenericRecordDescriptor =
    { grd_name       :: String
    , grd_arity      :: Int
    , grd_type_arity :: Int                    // arity of the type
    , grd_type       :: GenType                // type of the constructor
    , grd_fields     :: [String]
    }
```

The field descriptor is passed after **of** in the FIELD case of a generic function.

```
:: GenericFieldDescriptor =
    { gfd_name       :: String
    , gfd_index      :: Int                    // index of the field in the record
    , gfd_cons       :: GenericRecordDescriptor  // the record constructor
    }
```

Generic pretty printer.

```
generic gToString a ::        String    a            -> String
gToString {|Int|}             sep       x            = toString x
gToString {|UNIT|}            sep       x            = x
gToString {|PAIR|} fx fy      sep       (PAIR x y)   = fx sep x +++ sep +++ fy sep y
gToString {|EITHER|} fl fr    sep       (LEFT x)     = fl sep x
gToString {|EITHER|} fl fr    sep       (RIGHT x)    = fr sep x
gToString {|CONS of c|} fx    sep       (CONS x)
    | c.gcd_arity == 0
        = c.gcd_name
        = "(" +++ c.gcd_name +++ " " +++ fx " "  x +++ ")"
gToString {|RECORD of c|} fx  sep       (RECORD x)
       = "{" +++ c.grd_name +++ " | " +++ fx ", " x +++ "}"
gToString {|FIELD of f|} fx   sep       (FIELD x)    = f.gfd_name +++ "=" +++ fx x

toStr :: a -> String | gToString{|*|} a
toStr   x = gToString{|*|} "" x
```

7.6 Generic Functions and Uniqueness Typing

Uniqueness is very important in Clean. The generic extension can deal with uniqueness. The mechanism that derives generic types for different kinds is extended to deal with uniqueness information. Roughly speaking it deals with uniqueness attribute variables in the same way as it does with normal generic variables.

The type of standard mapping for lists with uniqueness

```
map :: (.a -> .b) ![.a] -> [.b]
```

Generic mapping with uniqueness. The instance on lists has the same uniqueness typing as the standard map

```
generic gMap a b :: .a -> .b
```

Uniqueness information specified in the generic function is used in typing of generated code.

Generated classes

```
class gMap{|*|} t             :: .t -> .t
class gMap{|*->*|} t          :: (.a -> .b) (.t .a) -> .t .b
class gMap{|*->*->*|} t       :: (.a1 -> .b1) (.a2 -> .b2) (.t .a1 .a2) -> .t .b1 .b2
```

Current limitations with uniqueness: generated types for higher order types require local uniqueness inequalities which are currently not supported.

Counter Example due to limitation in the current version of Clean.

```
class gMap{|(*->*)->*|} t  ::
    (A. (a:a) (b:b): (.a -> .b) -> (f:f a:a) -> g:g a:a, [f <= a, g <= b])
    (.t .f) -> .t .g
    , [f <= t, g <= t]
```

7.7 Generic Functions Using Other Generic Functions

Generic type variables may not be used in the context of a generic function, for example:

```
generic f1 a :: a -> Int | Eq a;        // incorrect
generic f2 a :: a -> Int | gEq{|*|} a; // incorrect
```

However generic function definitions that depend on other generic functions can be defined by adding a | followed by the required generic functions, separated by commas.

GenericDef	=	**generic** FunctionName TypeVariable+ [GenericDependencies] **::** FunctionType
GenericDependencies	=	**|** {FunctionName TypeVariable+ }-list

For example, to define `h` using `g1` and `g2`:

```
generic g1 a :: a -> Int;
generic g2 a :: a -> Int;
generic h a | g1 a, g2 a :: a -> Int;

h{|PAIR|} ha g1a g2a hb g1b g2b (PAIR a b)
    = g1a a+g2b b;
```

The algorithm for generating classes described in 7.2 is extended to add the dependent generic function arguments after each argument added by `(GenFun{|k|} b1 .. bn)`.

7.8 Exporting Generic Functions

Generic declarations and generic cases - both provided and derived - can be exported from a module. Exporting a generic function is done by giving the *generic* declaration in the DCL module. Exporting provided and derived generic cases is done by means of *derive*.

GenericExportDef	=	GenericDef **;**
	|	**derive** FunctionName {DeriveExportType [UsedGenericDependencies]}-list **;**
	|	**derive class** ClassName {DerivableType}-list **;**
	|	FunctionName **{|** GenericExportTypeArg **|}** {Pattern}+ **=** FunctionBody
GenericDef	=	**generic** FunctionName TypeVariable+ **::** FunctionType

DeriveExportType	=	TypeName
	|	GenericMarkerType [**of** UsedGenericInfoFields]
	|	PredefinedTypeConstructor
	|	TypeVariable
UsedGenericInfoFields	=	**{**[{FieldName}-list]**}**
	|	Variable
UsedGenericDependencies	=	**with** {UsedGenericDependency}
UsedGenericDependency	=	Variable
	|	**_**
GenericExportTypeArg	=	GenericMarkerType [**of** Pattern]
	|	**UNIT** | **PAIR** | **EITHER**

Example. Exporting of generic mapping. Definition as given in module GenMap.dcl

```
generic gMap a b :: .a -> .b
derive gMap c, PAIR, EITHER, CONS, FIELD, []
```

A generic function cannot be derived for an abstract data type, but it can be derived in the module where the abstract type defined. Thus, when one may export derived instance along with the abstract data type.

Additional information can be specified to make it possible for the compiler to optimize the generated code.

- The used generic info fields for generic instances of `OBJECT`, `CONS`, `RECORD` and `FIELD` can be specified by adding: `of` {`FieldName`}-list, at the end of the derive statement.

For example for:

```
gToString {|FIELD of {gfd_name}|} fx sep (FIELD x) = gfd_name +++ "=" +++ fx x
```

add: `of {gcd_name}` in the definition module:

```
derive gToString FIELD of {gfd_name}
```

and the function will be called with just a `gfd_name`, instead of a `GenericFieldDescriptor` record.

- Whether generic function dependencies for generic instances are used can be specified by adding: `with` followed by the list of dependencies. An _ indicates an unused dependency, a lower case identifier a (possibly) used dependency.

For example for:

```
generic h a | g1 a, g2 a :: a -> Int;
h {|OBJECT of {gtd_name}|} _ g1_a _ (OBJECT a) = g1_a gtd_name a;
```

Add: `with _ g1 _` in the definition module:

```
derive h OBJECT of {gtd_name} with _ g1 _;
```

`h` for `OBJECT` will be called without a function argument for `h` (for a of `OBJECT`), with `g1` and without `g2`, because `h` and `g2` are not used by the implementation.

- Generic cases for the generic representation types (`UNIT`, `PAIR`, `EITHER`, `OBJECT`, `CONS`, `RECORD`, `FIELD`) may be defined in definition modules (instead of a derive) using the same syntax as used in implementation modules.

Chapter 8

Dynamics

Dynamics are a new experimental feature of CLEAN. The concept is easy to understand, but the implementation is not so straightforward (see Vervoort and Plasmeijer, 2002). So, it will take some time before all bits and pieces are implemented and the system will work flawlessly. Please be patient.

What can you do with "Dynamics"? With "Dynamics" it is possible to store and exchange a CLEAN expression between (different) CLEAN applications in an easy and type-safe way. The expression may contain (unevaluated!) data and (unevaluated!) function applications. Here are some examples of its use.

- Almost all applications *store and fetch information to and from disk* (settings and the like). Traditionally, information written to file first has to be converted by the programmer to some (*String*) format. When the file is read in again a parser has to be constructed to parse the input and to convert the *String* back to the appropriate data structure. With `Dynamic`s one can store and retrieve (almost) any CLEAN data structure in a type-safe way with just one (!) function call. Not only data can be saved, but also code (unevaluated functions, higher order functions), which is part of the data structure being stored. `Dynamic`s make it easier to write a *persistent application*: an application that stores the settings in such a way that the next time the user will see everything in the same way as the last time the application was used.
- Different independently programmed CLEAN applications, even applications *distributed* across a network, can easily *communicate arbitrary expressions* that can contain data as well as code (unevaluated functions) in a type-safe way. `Dynamic`s can be communicated via files or via message passing primitives offered by the CLEAN libraries. The fact that CLEAN applications can communicate code means that a running CLEAN application can be extended with additional functionality. So, *plug-ins* and *mobile code* can be realized very easily and everything is *type-safe*.

To make all this possible we need some special facilities in the CLEAN language. But, in the CLEAN run-time system special support is needed as well. This includes *dynamic type checking*, *dynamic type unification*, dynamic encoding and *decoding* of arbitrary CLEAN *expressions and types*, *dynamic linking*, *garbage collection* of dynamics objects on disk, and *just-in-time code generation*.

In CLEAN 2.0 we have added a *dynamic type system* such that CLEAN now offers a *hybrid type system* with both *static* as well as *dynamic typing*. An object (expression) of static type can be packed into an object of dynamic type (a "`Dynamic`") and backwards. The type of a Dynamic can only be checked at run-time. An application can also check whether types of several Dynamics can be unified with each other, without a need to know what types are exactly stored into a Dynamic. In this way CLEAN applications can be used as control language to manage processes and plug-ins (see Van Weelden and Plasmeijer, 2002).

In this Chapter we first explain how a Dynamic can be constructed (see 8.1). In Section 8.2 we explain how the type of a `Dynamic` can be inspected via a pattern match, and how one can ensure that `Dynamics` fit together by using run-time type unification.

We explain in Section 8.3 how dynamics can be used to realize type safe communication of expressions between independently executing applications. In Section 8.4 we explain the basic architecture of the CLEAN run-time system that makes this all possible. Semantic restrictions and restrictions of the current implementation are summarized in Section 8.5.

8.1 Packing Expressions into a Dynamic

Since CLEAN is a strongly typed language (see Chapter 5), *every* expression in CLEAN has a *static type* determined at *compile time*. The CLEAN compiler is in general able to infer the static type of any expression or any function.

Example of CLEAN expressions and their static type:

```
3::Int
map::(a -> b) [a] -> [b]
map ((+) 1)::[Int] ->[Int]
MoveColorPoint Green::(Real,Real) -> ColorPoint
```

By using the keyword **dynamic** one can (in principle) change *any*expression of static type `::` `t` into a dynamically typed object of static type `::Dynamic`. Such a "dynamic" is an object (a record to be precise) that contains the original expression as well as an encoding of the original static type of the expression. Both the expression as well as the encoding of its static type, are packed into a dynamic. At run-time, the contents of a dynamic (the value of the expression as well the encoded type) can be inspected via a dynamic pattern match (see 8.2).

DynamicExpression = **dynamic** GraphExpr [`::` [UnivQuantVariables] Type [ClassContext]]

Example showing how one can pack an expression into a `Dynamic`. Optionally, the static type of the expression one wants to pack into a Dynamic can be specified.

```
dynamic 3
dynamic 3::Int
dynamic map::A.a b:(a->b) [a] -> [b]
dynamic map::(Int -> Real) [Int] -> [Real]
dynamic map ((+) 1)
dynamic MoveColorPoint Green
```

Example of a (constant) function creating a dynamic containing an expression of type `Tree Int`.

```
:: Tree a = Node a (Tree a) (Tree a) | Leaf

MyTree::Dynamic
MyTree = dynamic (DoubleTree 1 mytree)
where
    Doubletree rootvalue tree = Node rootvalue tree tree

    mytree = (Node 2 (Node 3 Leaf Leaf) Leaf)
```

Only the compiler is able to combine an expression with its type into a dynamic, such that it is guaranteed that the encoded type is indeed the type of the corresponding packed expression. However, as usual it is allowed to specify a more specific type than the one the compiler would infer. The compiler will check the correctness of such a (restricted) type specification. Polymorphic types can be stored as well.

- If an expression of polymorphic type is packed into a dynamic one needs to explicitly specify the universal quantifiers as well (see the example above).

In principle (there are a few exceptions), any algebraic data type can be packed into a dynamic, including basic types, function types, user defined types, polymorphic types, record types, all types of arrays, all types of lists, and existentially quantified types. The system can deal with synonym types as well. Restrictions hold for packing abstract data types, uniqueness types and overloaded functions, see the sub-sections below.

The static type of the object created with the keyword "dynamic" is the predefined type `Dynamic`. Since all objects created in this way are of type `Dynamic`, the compiler is in general not able anymore to determine the static type hidden in the `Dynamic` and it cannot check its type consistency anymore. The type stored into a `Dynamic` has to be checked at run-time (see 8.2 and 8.3).

8.1.1 Packing Abstract Data Types

Certain types simply cannot be packed into a `Dynamic` for fundamental reasons. Examples are objects of abstract predefined type that have a special meaning in the real world, such as objects of type `World` and of type `File`. It is not sound to pack objects of these types into a Dynamic, because their real world counterpart cannot be packed and stored.

- Abstract data types that have a meaning in the real world (such as World, File) cannot be packed into Dynamic.

A compile time error message will be given in those cases were the compiler refuses to pack an expression into a `Dynamic`.

In the current implementation there are additional restrictions on the kind of types that can be packed into a`Dynamic`. Currently it is not possible to pack any abstract data type into a dynamic at all. The reason is that the test on equality of abstract types is not easy: it does not seem to be enough to test the equality of the definitions of the types involved. We should also test whether the operations defined on these abstract data types are exactly the same. The most straightforward way to do this would be to require that abstract data types are coming from the same module (repository).

- Expressions containing objects of abstract data type cannot be packed into a Dynamic. We are working on this. Check the latest version of the `Clean` system.

8.1.2 Packing Overloaded Functions

Overloaded functions can also be packed into a `Dynamic`. In that case the corresponding type classes (see 6.1) are packed as additional dictionary argument of the function into the `Dynamic`. When the `Dynamic` is unpacked in a pattern match, the same type classes have to be defined in the receiving function, otherwise the pattern match will fail (see 8.2.2).

Example: storing an overloaded function into a `Dynamic`.

```
OverloadedDynamic:: Dynamic
OverloadedDynamic = dynamic plus :: A.a:a a -> a | + a
where
    plus:: a a -> a | + a
    plus x y = x + y
```

Currently, when an overloaded function is packed into a dynamic, one explicitly has to specify the type, including the forall quantifier and the class context.

8.1.3 Packing Expressions of Unique Type

Expressions of unique type (see Chapter 9) can also be packed into a Dynamic. However, the run-time system cannot deal with uniqueness type variables or with coercion statements (attribute variable inequalities). One can only use the type attribute "`*`". Furthermore, one has to explicitly define the desired unicity in the type of the expression to be packed into a `Dynamic`. Otherwise the unicity properties of the packed expression will not be stored. As usual, the compiler will check whether the specified type (including the uniqueness type attributes) of the packed expression is correct.

Example: packing a function into a `Dynamic` that can write a character to a unique file.

```
MyDynamic:: Dynamic
MyDynamic = dynamic fwritec :: Char *File -> *File
```

- Uniqueness type variables and coercion statements cannot be part of a type packed into a Dynamic.
- Uniqueness type attributes are only taken into account if they are explicitly specified in the type of the packed dynamic.

Counter Example: `Dynamics` cannot deal with uniqueness type variables or with attribute variable inequalities.

```
MyDynamic:: Dynamic
MyDynamic = dynamic append :: [.a] u:[.a] -> v:[.a]         , [u<=v]
```

8.1.4 Packing Arguments of Unknown Type

The compiler is not in all cases capable to infer the concrete type to be assigned to a `Dynamic`. For instance, when a polymorphic function is defined it is in general unknown what the type of the actual argument will be. If it is polymorphic, it can be of any type.

- An argument of polymorphic type cannot be packed into a Dynamic.

Counter Example of a function creating a `Dynamic`. Arguments of polymorphic type cannot be packed into a Dynamic.

```
WrongCreateDynamic:: t -> Dynamic
WrongCreateDynamic any = dynamic any
```

If one wants to define a function that can wrap an arbitrary argument into a `Dynamic`, not only the value, but also the concrete static type of that argument has to be passed to the function. For efficiency reasons, we of course do not want to pass the types of all arguments to all functions. Therefore, we have to know which of the arguments might be packed into a `Dynamic`. A special class context restriction is used to express this. So, instead of a *polymorphic* function one has to define an *overloaded* function (see Chapter 6). The class `TC` (for `T`ype `C`ode) is predefined and an instance of this class is required whenever an argument of unknown type is packed into a dynamic. The compiler will automatically generate an appropriate instance of the TC when needed.

Example of an overloaded function that can wrap an argument of arbitrary type into a `Dynamic`.

```
CreateDynamic:: t -> Dynamic | TC t
CreateDynamic any = dynamic any

MyTree:: Dynamic
MyTree = CreateDynamic (Node 2 (Node 3 Leaf Leaf) Leaf)
```

8.1.5 Using Dynamic Typing to Defeat the Static Type System

Dynamic typing can also be used to do things the static type system would forbid. For instance, lists require that all lists elements are of the same type. Since all dynamic expressions are of type `Dynamic`, one can combine objects of static different type into a list by turning them into a `Dynamic`.

Example: three ways to pack objects of different type into a list. The first method is to define a new type in which all types one likes to pack into a list are summarized with an appropriate constructor to distinguish them. For unpacking one can make a case distinction on the different constructors in a pattern match. Everything is nice statically typed but one can only pack and unpack the types that are mentioned in the wrapper type.

```
:: WrapperType = I Int | R Real | C Char

MyWrapperList = [I 1, R 3.14, C 'a']
```

The next way to pack objects of different types is by defining a list structure using an existential type (see 5.1.3). Any type can be packed now but the disadvantage is that there is no simple way to distinguish the elements and unpack them once they are packed.

```
:: ExstList = E.a: Cons a ExstList | Nil

MyExstList = Cons 1 (Cons 3.14 (Cons 'a' Nil)
```

The third way is to wrap the values into a Dynamic. Any type can be packed and via a pattern match one can unwrap them as well. It is very inefficient though and one can only unwrap a value by explicitly naming the type in the pattern match (see 8.2).

```
MyDynamicList = [dynamic 1, dynamic 3.14, dynamic 'a']
```

It is possible to write CLEAN programs in which all arguments are dynamically typed. You can do it, but it is not a good thing to do: programs with dynamics are less reliable (run-time type errors might occur) and much more inefficient. `Dynamics` should be used for the purpose they are designed for: type-safe storage, retrieval, and communication of arbitrary expressions between (independent) applications.

8.2 Unpacking Dynamics Using a Dynamic Pattern Match

When a `Dynamic` is created (see above), its static type is the type `Dynamic`. The compiler is in general not able anymore to determine what the original type was of the expression that is packed into a `Dynamic`. The only way to figure that out is at run-time (that why it is called a `Dynamic`), by inspecting the dynamic via a pattern match (see 3.2) or a case construct (see 3.4.2). With a pattern match on a Dynamic one cannot only inspect the *value* of the expression that was packed into a `Dynamic`, but also its original *type*. So, with `Dynamics` run-time type checking and dynamic type unification is possible in CLEAN with all the advantages (more expressions can be typed) and disadvantages (type checking may fail at run-time) of a dynamic type system. The programmer has to take care of handling the case in which the pattern match fails due to a non-matching dynamic type.

```
DynamicPattern          =  (GraphPattern :: DynamicType)
DynamicType             =  [UnivQuantVariables] {DynPatternType}+ [ClassContext]
DynPatternType          =  Type
                        |  TypePatternVariable
                        |  OverloadedTypePatternVariable
TypePatternVariable     =  Variable
OverloadedTypeVariable  =  Variable^
```

Any expression that can be packed into a dynamic can also be unpacked using a dynamic pattern match. With a pattern match on `Dynamic`s a case distinction can be made on the contents of the `Dynamic`. If the actual `Dynamic` matches the type *and* the value specified in the pattern, the corresponding function alternative is chosen. Since it is known in that case that the `Dynamic` matches the specified type, this knowledge is used by the static type system: dynamics of known type can be handled as ordinary expressions in the body of the function. In this way dynamics can be converted back to ordinary statically typed expressions again. The static type checker will use the knowledge to check the type consistency in the body of the corresponding function alternative.

Example: Use of a dynamic pattern match to check whether a `Dynamic` is of a specific predefined type. The first alternative of the function transform matches if the `Dynamic` contains the Integer of value `0`. The second alternative is chosen if the `Dynamic` contains any Integer (other than `0`). The third alternative demands a function from `[Int]` to `[Int]`. The next alternative is chosen if the `Dynamic` is a pair of two `[Int]`. If none of the alternatives match, the last alternative is chosen. The program will yield an empty list in that case.

```
transform :: Dynamic -> [Int]
transform (0 :: Int)                = []
transform (n :: Int)                = [n]
transform (f :: [Int]->[Int])       = f [1..100]
transform ((x,y) :: ([Int],[Int])) = x ++ y
transform other                     = []
```

Warning: when defining a pattern match on `Dynamics`, one should always be aware that the pattern match might fail. So, we advise you to *always*include an alternative that can handle non-matching dynamics. The application will otherwise abort with an error message that none of the function alternatives matched.

Example: use of a dynamic pattern match to check whether a Dynamic is of a specific user defined algebraic data type. If the Dynamic contains a Tree of Int, the function CountDynamicLeafs will count the number of leafs in this tree. Otherwise CountDynamicLeafs will return 0.

```
:: Tree a = Node a (Tree a) (Tree a) | Leaf

CountDynamicLeafs :: Dynamic -> Int
CountDynamicLeafs (tree :: Tree Int)  = countleafs tree
CountDynamicLeafs other               = 0
where
    countleafs :: (Tree Int) -> Int
    countleafs  tree = count tree 0
    where
        count:: (Tree a) Int -> Int
        count Leaf nleafs              = nleafs + 1
        count (Node left right) nleafs = count left (count right nleafs)

MyTree :: Dynamic
MyTree = dynamic (Node 1 (Node 2 (Node 3 Leaf Leaf) Leaf) (Node 4 Leaf Leaf))

Start :: Int
Start = CountDynamicLeafs MyTree
```

Example: use of a dynamic pattern match to check whether a Dynamic is a polymorphic function (the identity function in this case).

```
TestId :: Dynamic a -> a
TestId (id :: A.b: b -> b) x = id x
TestId else x                = x
```

- To avoid confusion with type pattern variables (see 8.2.4 and 8.2.5), polymorphic type variables have to be explicitly introduced with the forall quantifier (A.).
- Quantifiers are only allowed on the outermost level (Rank 1).

Dynamics can be created by functions in other modules or even come from other (totally different) Clean applications (see 8.3). It is therefore possible that in a Dynamic a type with a certain name is stored, yet this type might have a type definition which is (slightly or totally) different from the type known in the matching function. So, the context in which a dynamic is packed might be totally different from the context in which the dynamic is unpacked via a pattern match. Hence, it is not enough that matching type constructors have identical names; they should also have exactly the same type definition. If not, the match will fail.

Two types are considered to be equal if and only if all the type definitions (type constructors, data constructors, class definitions) are syntactically identical modulo the names of type variables (alpha conversion is allowed). Type equivalence of type constructors is automatically checked by the Clean run-time system, even if these types are defined in totally different Clean applications. To make this possible, we had to change the architecture of the Clean run-time system (see 8.4).

So, when a pattern match on a dynamic takes place, the following things are checked in the indicated order (case constructs are handled in a similar way):

1) All the type constructors (either of basic type, predefined or user defined) specified in a dynamic pattern will be compared with the name of the corresponding actual type constructors stored in the dynamics. If corresponding type constructors have different names, the pattern match fails and the next alternative is tried.

2) If in the pattern match, corresponding type's constructors have the same name, the run-time system will check whether their type definitions (their type might have been defined in different Clean applications or different Clean modules) are the same as well. The system knows where to find these type definitions (see 8.3 and 8.4). If the definitions are not the same, the types are considered to be different. The pattern match fails and the next alternative is tried.

3) If the types are the same, the actual data constructors (constant values) are compared with the data constructors specified in the patterns, as usual in a standard pattern match (see 3.2) without dynamics. If all the specified constants match the actual values, the match is successful and the corresponding function alternative is chosen. Otherwise, the pattern match fails and the next alternative is tried.

In the current implementation there are additional restrictions on the kind of types that can be packed into a `Dynamic` and therefore also be unpacked from a `Dynamic` (see 8.2.1, 8.2.2, and 8.2.3).

In a dynamic pattern match one can explicitly specify to match on a certain type constructors (e.g. `Tree Int`). One can also use a type pattern variable (see 8.2.4) to specify a type scheme with which the actual type has to be unified. By using overloaded variables (see 8.2.5) defined in the type of the function, the static context in which a function is used can have influence on the kind of dynamic that is accepted. So, there are two special types of variables that can occur in a type pattern: type pattern variables and overloaded type pattern variables.

8.2.1 Unpacking Abstract Data Types

- It is not yet possible to pack or unpack an abstract data type. See also 8.1.1.

8.2.2 Unpacking of Overloaded Functions

One can specify a class restriction in the type in a dynamic pattern match. The system will check whether the actual dynamic contains a function that is indeed overloaded with exactly the same class context restriction as specified in the pattern. Two class definitions are regarded to be equal if all involved class definitions are syntactically equal, modulo alpha conversion of the type variables.

- One is obligated for overloaded type variables to introduce them via the forall quantifier in the pattern, to avoid confusion with type pattern variables (see 8.2.4 and 8.2.5).
- Quantifiers are only allowed on the outermost level.

Example: Unpacking of an overloaded function. The pattern match will only be successful if the dynamic contains a function overloaded in +. The corresponding class definitions will be checked: the definition of the class + has to be the same as the class + known in the context where the dynamic has been created. Due to the application of `plus 2 3`, the type checker will require an instance for + on integer values.

```
CheckDynamic:: Dynamic -> Int
CheckDynamic (plus :: A.a : a a -> a | + a) = plus 2 3
CheckDynamic else                           = 0
```

8.2.3 Unpacking Expressions of Unique Type

Expressions of unique type (see Chapter 9) can also be unpacked via a dynamic pattern match. However, the run-time system cannot deal with uniqueness type variables or with coercion statements (attribute variable inequalities). One can only use the type attribute "*". The match will only be successful if the specified types match and all type attributes also match. No coercion from unique to non-unique or the other way around will take place.

Example: Unpacking a function that can write a character to a unique file.

```
WriteCharDynamic:: Dynamic Char *File -> *File
WriteCharDynamic (fwc :: Char *File -> *File) char myfile = fwc char myfile
WriteCharDynamic else char myfile                         = myfile
```

- Uniqueness type variables and coercion statements cannot be used in a dynamic pattern match.
- The type attributes of the formal argument and the actual argument have to be exactly the same. No coercion from unique to non-unique or the other way around will take place.

8.2.4 Checking and Unifying Types Schemes using Type Pattern Variables

In an ordinary pattern match one can use *data constructors* (to test whether an argument is of a *specific value*) and *variables* (which match on *any concrete value* in the domain). Similarly, in a pattern match on a dynamic type one can use *type constructors* (to test whether a dynamic contains an expression of a *specific type*) and *type pattern variables* (which match on *any type*). However, there are differences between ordinary variables and type pattern variables as well. All ordinary variable symbols introduced at the left-hand side of a function definition must have different names (see 3.2). But, the same type variable symbol can be used several times in the left-hand side (and, of course, also in the right-hand side) of a function definition. Type pattern variables have the function alternative as scope and can appear in a pattern as well as in the right-hand side of a function in the context of a dynamic type.

TypePatternVariable = Variable

Each time the *same* type variable is used in the left-hand side, the pattern matching mechanism will try to *unify* the type variable with the concrete type stored in the dynamic. If the same type variable is used several times on the left hand-side, the most general unifier is determined that matches on *all* corresponding types in the dynamics. If no general unifier can be found, the match will fail. As usual, of all corresponding type constructors it will be checked whether they are indeed the same: the corresponding type definitions have to be equivalent (see 8.2). Type equivalence of matching type constructors is automatically checked by the `Clean` run-time system, even if these types are defined in different `Clean` applications.

Type pattern variables are very handy. They can be used to check for certain type schemes or to check the internal type consistency between different Dynamics, while the checking function does not exactly has to know which concrete types are actually stored in a dynamic. One can make use of type pattern variables to manage and control plug-ins in a flexible way see 8.3).

The function `dynApply` has two arguments of type `Dynamic` and yields a value of type `Dynamic` as well. The first `Dynamic` has to contain a function unifiable with type `(a -> b)`, the second argument has to be *unifiable* with the argument type `a` the function is expecting. In this way we can ensure that it is type technically safe to apply the function to the argument, without exactly knowing what the actual types are. The result will have the statically unknown type `b`, but, by packing this result into a Dynamic again, the static type system is happy: it is a `Dynamic`. If the dynamics types cannot be unified, it is not type safe to apply the function to the argument, and the next alternative of `dynApply` is chosen. It yields an error message stored into a dynamic.

```
dynApply :: Dynamic Dynamic -> Dynamic
dynApply (f :: a -> b) (x :: a)     = dynamic (f x :: b)
dynApply  df            dx          = dynamic ("cannot apply ",df," to ",dx)

Start = dynApply (dynamic (map ((+) 1)) (dynamic [1..10])
```

Type pattern variables behave similar as existentially quantified type variables (see 5.1.3). It is statically impossible to determine what the actual type of a type pattern variable is. So, in the static type system one cannot define an expression which type is depending on the value of a type pattern variable. The static type system cannot deal with it, since it does no know its value. However, an expression which type is depending on the type assigned to a type pattern variable can be packed into a dynamic again, because this is done at run-time. See the `dynAppy` example above.

- It is not allowed to create an expression which static type is depending on the value of a type pattern variable.

Counter Example: It is not possible to let a static type depend on the value of a type pattern variable. The actual value of the type `b` in `WrongDynApply` is unknown at run-time. This example will result into a type error. See 8.2.5 for a legal variant of this function.

```
WrongDynApply :: Dynamic Dynamic -> ???
WrongDynApply (f :: a -> b) (x :: a)  = f x
WrongDynApply df            dx        = abort "cannot perform the dyanmic application"

Start = WrongDynApply (dynamic (map ((+) 1)) (dynamic [1..10]) ++ [11..99]
```

Note: don't confuse type pattern variables with the other variables that can appear in a dynamic type to indicate polymorphic or overloaded functions or constructors. The latter are introduced via a quantifier in the type. See also 8.2.5.

8.2.5 Checking and Unifying Unknown Types using Overloaded Type Variables

In a dynamic pattern match one can explicitly state what type of a `Dynamic` is demanded. But it is also possible to let the static context in which a function is used impose restrictions on the Dynamic to be accepted. This can be realized by using *overloaded type variables* in a dynamic pattern. Such an overloaded type variable has to be introduced in the type of the function itself and the variable should have the predefined type class `TC` (see 8.1.1) as context restriction. This makes it possible to let the overloading mechanism determine what the demanded type of a Dynamic has to be (the "type dependent functions" as introduced by Marco Pil, 1999). By using such an overloaded type variable in a dynamic pattern, the type assigned by the static overloading mechanism to this variable is used as specification of the required type in the dynamic pattern match. The caret symbol (^) is used as suffix of a type pattern variable in a dynamic pattern match to indicate that an overloaded type variable is used, instead of a type pattern variable. Overloaded type variables have the whole function definition as scope, including the type of the function itself.

```
OverloadedTypeVariable  =  Variable^
```

- An overloaded type pattern variable has to be introduced in the type definition of the function.
- The predefined type class `TC` (see 8.1.1) has to be specified as context restriction on the global type pattern variable.
- As is usual with overloading (see 6.6), in some cases the compiler is not able to resolve overloading, e.g. due to internally ambiguously overloading.

Example: The function `Start` appends `[11..99]` to the result of `FlexDynApply`. So, it is clear that `FlexDynApply` will have to deliver a `[Int]` to the function `Start`. The additional context restriction `TC b` turns `FlexDynApply into an overloaded function in b`. The function `FlexDynApply` will not deliver *some* dynamic type `b`, but *the* static type `b` that is demanded by the context applying `FlexDynApply`. The overloading mechanism will automatically pass as additional parameter information about the static type `b` that is required by the context. This type is then used to check the actual type of the dynamic in the dynamic pattern match

```
FlexDynApply :: Dynamic Dynamic -> b | TC b
FlexDynApply (f :: a -> b^) (x :: a)   = f x
FlexDynApply df             dx         = abort "cannot perform the dyanmic application"

Start = FlexDynApply (dynamic (map ((+) 1)) (dynamic [1..10]) ++ [11..99]
```

Example: The function `lookup` will look up a value of a certain type `a` in its lists of `Dynamic`s. The type it will search for depends on the context in which the function `lookup` is used. In Start the lookup function is used twice. In the first case an integer value is demanded (due to `+ 5`), in the second case a real value (due to `+ 2.5`) is required. The program will be aborted if a value of the required type cannot be found in the list.

```
lookup :: [Dynamic] -> a  | TC a
lookup [(x :: a^):xs] = x
lookup [x:xs]         = lookup xs
lookup []             = abort "dynamic type error"

Start = (lookup DynamicList + 5, lookup DynamicList + 2.5)  // result will be (6,5.64)

DynamicList = [dynamic 1, dynamic 3.14, dynamic 'a']
```

Note: don't confuse overloaded type variables with type pattern variables or the other variables that can appear in a dynamic type to indicate polymorphic or overloaded functions or constructors. The latter are introduced via a quantifier in the type.

Example: the following artificial example the kinds of type variables that can be used in a dynamic pattern are shown. In the first alternative a type variable `a` is used (introduced by the forall quantifier). This alternative only matches on a polymorphic function. In the second alternative an overloaded type variable is used (indicated by `a^`) referring to the overloaded type variable `a | TC a` introduced in the function body. It will match on a function of the same type as the actual type of the second argument of AllSortsOfVariables. The last alternative uses type pattern variables `a` and `b`. It matches on any function type, although this function is not used.

```
AllSortsOfVariables :: Dynamic  a -> a | TC a
AllSortsOfVariables (id::A.a : (a -> a)) x  = id x
AllSortsOfVariables (f::a^ -> a^)        x  = f x
AllSortsOfVariables (f::a -> b)          x  = x
```

8.3 Type Safe Communication using Dynamics

As explained in the introduction of this Chapter, the most important practical use of `Dynamic`s is enabling type safe communication of data and code between different (distributed) CLEAN applications. When a `Dynamic` is stored or communicated it will be encoded (serialized) to a string. So, in principle almost any communication media can be used to communicate `Dynamic`s. In this section we only explain how to store and retrieve a `Dynamic` from a `File`. It is also possible to communicate a `Dynamic` directly via a channel or via send / receive communication primitives. The actual possibilities are depending on the facilities offered by CLEAN libraries. This is outside the scope of this CLEAN language report.

If a CLEAN application stores a `Dynamic` into a `File` *any* other (totally different) CLEAN application can read the `Dynamic` from it. Since a `Dynamic` can contain data as well as code (unevaluated function applications), this means that any part of one CLEAN program can be plugged into another. Since CLEAN is using compiled code, this has a high impact on the run-time system (see 8.4).

One can read and write a `Dynamic` with just *one* function call. In the CLEAN library `StdDynamic` the functions `readDynamic` and `writeDynamic` are predefined. As usual in CLEAN, uniqueness typing is used for performing I/O (see 9.1). When a `Dynamic`is written, the whole expression (the graph expression and its static type) is encoded symbolically to a `String` and stored on disk. When a `Dynamic` is read, it is read in lazily. Only when the evaluation of the `Dynamic` is demanded (which can only happen after a successful pattern match), the String is decoded back to a `Dynamic`. If new function definitions have to be plugged in, this will be done automatically (see 8.4). This lazy reading is also done for `Dynamics` stored into a `Dynamic`. So, a `Dynamic` can only be plugged in if its type is approved in a `Dynamic` pattern match.

Standard functions for reading and writing of a Dynamic.

```
definition module StdDynamic

...

writeDynamic :: Dynamic String *World -> *(Bool,*World)

readDynamic :: String *World -> *(Bool, Dynamic, *World)
```

The use of `Dynamics` is shown in the examples below. Each example is a complete CLEAN application.

Example of a CLEAN application writing a `Dynamic` containing a value of type `Tree Int` to a `File` named `DynTreeValue`. This example shows that data can be stored to disk using the CLEAN function `writeDynamic`.

```
module TreeValue

import StdDynamic, StdEnv

:: Tree a = Node a (Tree a) (Tree a) | Leaf

Start world
# (ok,world) = writeDynamic "DynTreeValue" MyTree world
| not ok     = abort "could not write MyTree to file named DynTreeValue"
| otherwise  = world
where
    MyTree::Dynamic
    MyTree = dynamic (Node 1 mytree mytree)
    where
        mytree = (Node 2 (Node 3 Leaf Leaf) Leaf)
```

Example of another CLEAN application writing a `Dynamic` containing the function countleafs to a File named CountsLeafsinTrees. This function can count the numbers of leafs in a `Tree` and is of type `(Tree Int) -> Int`. This examples shows that code (in this case the function `CountLeafs`) can be stored on disk as well, just using `WriteDynamic`.

```
module CountLeafs

import StdDynamic, StdEnv

:: Tree a = Node a (Tree a) (Tree a) | Leaf

Start world
# (ok,world) = writeDynamic "CountsLeafsinTrees" CountLeafs world
| not ok     = abort "could not write dynamic"
| otherwise  = world
where
    CountLeafs = dynamic countleafs

    countleafs:: (Tree Int) -> Int
    countleafs  tree = count tree 0
    where
        count:: (Tree a) Int -> Int
        count Leaf nleafs                = nleafs + 1
        count (Node left right) nleafs   = count left (count right nleafs)
        count else                       = abort "count does not match"
```

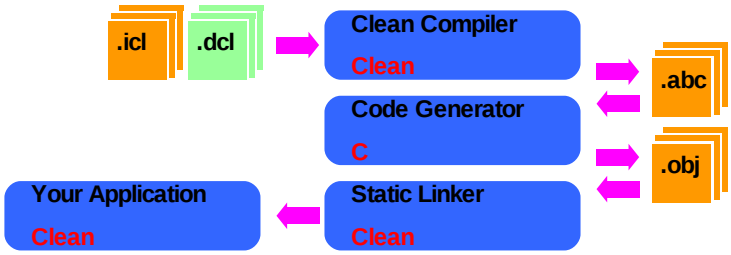
The third CLEAN application reads in the file TreeValue containing a `Tree Int` and the function `countleafs` (a plugin) that can counts the number of `Leaf`s in a `Tree`. So, new functionality is added to the running application Apply. By using the function `dynapply` the new plugged in function countleafs is applied to the tree that has been read in as well. The application `Apply` itself has a function to count the number of nodes and applies this function on the tree read in.
Note that this application will only work if all the type `Tree`s defined in the different applications are exactly the same (module the names for the type variables used).

```
module Apply

import StdDynamic, StdEnv

:: Tree a = Node a (Tree a) (Tree a) | Leaf

Start world
# (ok,countleafs,world)    = read "CountsLeafsinTrees" world
| not ok                   = abort ("could not read CountsLeafsinTrees")
# (ok,treevalue,world)     = read "TreeValue" world
| not ok                   = abort ("could not read TreeValue")
| otherwise  =    (    countnodes (case treevalue of (v::(Tree Int))= v) 0
                 ,    dynapply countleafs treevalue
                 )
where
    dynapply :: Dynamic Dynamic -> Dynamic
    dynapply (f::a -> b) (v::a)   = dynamic (f v)
    dynapply df          dv       = dynamic "incorrectly typed dynamic application"

    countnodes Leaf nnodes               = nnodes
    countnodes (Node left right) nnodes = countnodes left (countnodes right (nnodes+1))
```

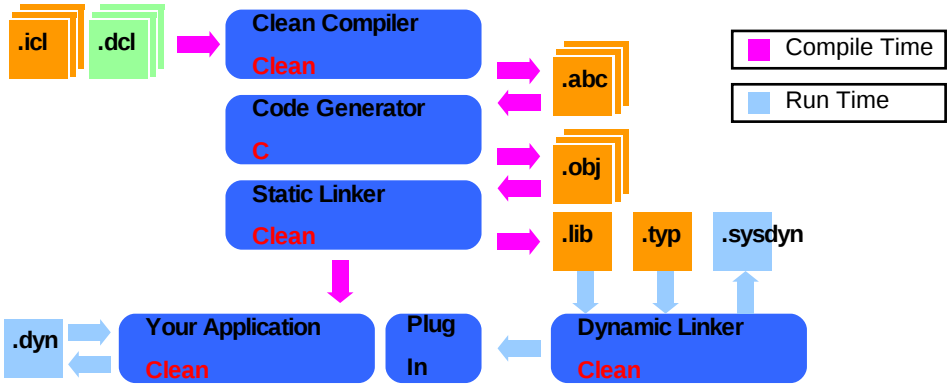
8.4 Architecture of the implementation

From the examples above we learn that a `Dynamic` stored on disk can contain data as well as code (unevaluated functions and function applications). How is this information stored into a file and how is a running CLEAN application extended with new data and new functionality? To understand this one has to know a little bit more about how a CLEAN application is generated.

CLEAN applications are not interpreted via an interpreter. Executables are generated using compiled machine code. Storing a `Dynamic` into disk and retrieving it again from disk cannot simply be done by (re) interpretation of CLEAN source code.



The CLEAN compiler (written in CLEAN) compiles CLEAN implementation modules (.icl and .dcl files) to machine independent abc-code (.abc files). The CLEAN definition modules are used to check the type consistency between the separately programmed CLEAN modules. The abc-code contains machine instructions for a virtual abstract machine, the abc-machine (see Plasmeijer and van van Eekelen, 1993). The abc-code is a kind of platform independent byte code specially designed for CLEAN. The Code Generator (the one and only application in the Clean system which is written in C) translates the abc-code into platform dependent symbolic machine code (.obj files under Windows). The code generator can generate code for different platforms such as for Intel (Windows, Linux), Motorola (Mac) and Sparc (Unix) processors. The Static Linker (written in CLEAN) links all the object modules of one CLEAN program together into a click able executable application (.exe file under Windows). The compilation scheme described above can be used even if Dynamics are internally used in an application. But, as soon as `Dynamic`s are communicated to `File` or communicated to another program, a different run-time support is needed and the traditional compilation scheme is changed to prepare this.



In the changed compilation scheme, the static linker not only generates an application (actually, currently it is a .bat file), but it also generates two additional files. One is called the code repository (.lib file). All object codes of your application are collected here. The other file (.typ file) is a repository that contains all type definitions. The repositories serve as a database which is accessed by the Dynamic Linker. Whenever an (other) running application needs code for a plug in or a type that has to be checked, the Dynamic Linker will look it up in the appropriate repository. Each time a CLEAN program is recompiled, new repositories are created. Repositories should not be removed by hand because it might make the `Dynamic`s stored on disk unreadable. A special garbage collector is provided that can remove unused repositories.

When a CLEAN application doing dynamic I/O is started, a special linker (the Dynamic Linker, written in CLEAN) is started with it as well (if it is not already running). The Dynamic Linker is a server application on the computer. It will serve all running CLEAN programs that read or write `Dynamic`s. The Dynamic Linker will construct the application or plug-in at run-time in the same way as the Static Linker would do at compile-time for a conventional CLEAN program. There is no efficiency penalty once the code is linked in.

When a `Dynamic` is written to disk using the function `writeDynamic`, two (!) files are created: a .dyn file and a .sysdyn file. The .sysdyn file contains the actual information: a String encoding of the dynamic. This sysdyn file is used by the Dynamic Linker and should not be touched by the user because it might make the `Dynamic`s stored on disk unreadable. The special garbage collector provided will also remove unused .sysdyn files.

The user may only touch and use the .dyn file that contains references to the actual dynamic stored in the .sysdyn file. The .dyn file can be seen as a "typed" file. It can be handled like any other user file. It can be renamed, moved or deleted without any danger.

When a `Dynamic`is written to a file, an encoding of the graph and its type are written to disk. The graph is encoded in such a way that sharing will be maintained when the graph is read in again. The stored graph may contain unevaluated functions. In a `Dynamic` on disk, functions are represented by symbolic pointers to the corresponding code repositories. The types stored in a Dynamic on disk point to the corresponding type definitions stored in the type repositories.

No plug-in will be plugged in unless its type is approved. When a `Dynamic` stored on disk is read in by an (other) application, first only a pointer to the stored `Dynamic`is used in the CLEAN program. To use a Dynamic in an application one first has to examine it in a pattern match. In the pattern match the type of the `Dynamic` is unified with a specified type or with the type of another `Dynamic`s. If the run-time type unification is successful, the Dynamic Linker will check whether all type definitions of the types involved are identical as well. This type information is fetched from the corresponding type repository when needed. If the types are identical and the conventional pattern match succeeds as well, the corresponding function body can be evaluated. Only when the evaluation of the stored Dynamic is demanded, the graph encoded in the Dynamic on disk is reconstructed as far as needed (`Dynamic`s nested in a `Dynamic` are reconstructed lazily). The Dynamic Linker links in the code corresponding to the unevaluated functions stored in the `Dynamic`. It is fetched from the code repository and plugged in the running (!) application. In some cases the Dynamic Linker will ask the Code Generator to generate new code (just-in-time code generation) to construct the required image. The Dynamic Linker has to ensure that identical data types defined in different applications are linked in such a way that they are indeed considered to be of the same type at run-time.

8.5 Semantic Restrictions on Dynamics

- The following types cannot be packed/unpacked: abstract data types, uniqueness types, overloaded types. We are working on it.

Chapter 9

Uniqueness Typing

Although CLEAN is purely functional, operations with side-effects (I/O operations, for instance) are permitted. To achieve this without violating the semantics, the classical types are supplied with so called uniqueness attributes. If an argument of a function is indicated as unique, it is guaranteed that at run-time the corresponding actual object is local, i.e. there are no other references to it. Clearly, a destructive update of such a "unique object" can be performed safely.

The uniqueness type system makes it possible to define direct interfaces with an operating system, a file system (updating persistent data), with GUI's libraries, it allows to create arrays, records or user defined data structures that can be updated destructively. The time and space behavior of a functional program therefore greatly benefits from the uniqueness typing.

Uniqueness types are deduced automatically. Type attributes are polymorphic: attribute variables and inequalities on these variables can be used to indicate relations between and restrictions on the corresponding concrete attribute values.

Sometimes the inferred type attributes give some extra information on the run-time behavior of a function. The uniqueness type system is a transparent extension of classical typing that means that if one is not interested in the uniqueness information one can simply ignore it.

Since the uniqueness typing is a rather complex matter we explain this type system and the motivation behind it in more detail. The first Section (9.1) explains the basic motivation for and ideas behind uniqueness typing. Section 9.2 focuses on the so-called uniqueness propagation property of (algebraic) type constructors. Then we show how new data structures can be defined containing unique objects (Section 9.3). Sharing may destroy locality properties of objects. In Section 9.4 we describe the effect of sharing on uniqueness types. In order to maintain referential transparency, it appears that function types have to be treated specially. The last Section (9.5) describes the combination of uniqueness typing and overloading. Especially, the subsections on constructor classes and higher-oder type definitions are very complex: we suggest that the reader skips these sections at first instance.

9.1 Basic Ideas behind Uniqueness Typing

The *uniqueness typing* is an extension of classical Milner/Mycroft typing. In the uniqueness type system *uniqueness type attributes* are attached to the classical types. Uniqueness type attributes appear in the *type specifications of functions* see 9.4) but are also permitted in the definitions of *new data types* (see 9.3). A classical type can be prefixed by one of the following uniqueness type attributes:

```
Type           =  {BrackType}+
BrackType      =  [Strict] [UnqTypeAttrib] SimpleType
UnqTypeAttrib  =  *                      // type attribute "unique"
               |  UniqueTypeVariable :   // a type attribute variable
               |  .                      // an anonymous type attribute variable
```

The basic idea behind uniqueness typing is the following. Suppose a function, say `F`, has a unique argument (an argument with type `*σ`, for some `σ`). This attribute imposes an additional restriction on applications of `F`.

It is *guaranteed* that `F` will have private ("unique") access to this particular argument (see Barendsen and Smetsers, 1993; Plasmeijer and Van Eekelen, 1993): the object will have a reference count of 1 *at the moment* it is inspected by the function. It is important to know that there can be more than 1 reference to the object before this specific access takes place. If a uniquely typed argument is not used to construct the function result it will become garbage (the reference has dropped to zero). Due to the fact that this analysis is performed statically the object can be garbage collected (see Chapter 1) at compile-time. It is harmless to reuse the space occupied by the argument to create the function result. In other words: *it is allowed to update the unique object destructively without any consequences for referential transparency*.

Example: the I/O library function `fwritec`is used to write a character to a file yielding a new file as result. In general it is semantically not allowed to overwrite the argument file with the given character to construct the resulting file. However, by demanding the argument file to be unique by specifying

```
fwritec:: Char *File -> *File
```

It is guaranteed by the type system that `fwritec` has private access to the file such that overwriting the file can be done without violating the functional semantics of the program. The resulting file is unique as well and can therefore be passed as continuation to another call of e.g. `fwritec` to make further writing possible.

```
WriteABC:: *File -> *File
WriteABC file = fwritec 'c' (fwritec 'b' (fwritec 'a' file))
```

Observe that a unique file is passed in a single threaded way (as a kind of unique token) from one function to another where each function can safely modify the file knowing that is has private access to that file.

One can make these intermediate files more vissible by by writing the `WriteABC` as follows.

```
WriteABC file = file3
where
    file1 = fwritec 'a' file
    file2 = fwritec 'b' file1
    file3 = fwritec 'c' file2
```

or, alternatively (to avoid the explicit numbering of the files),

```
WriteABC file
    #   file = fwritec 'a' file
        file = fwritec 'b' file
    =   fwritec 'c' file
```

The type system makes it possible to make no distinction between a CLEAN file and a physical file of the real world: file I/O can be treated as efficiently as in imperative languages. The uniqueness typing prevents writing while other readers/writers are active. E.g. one cannot apply `fwritec` to a file being used elsewhere.

For instance, the following expression is *not* approved by the type system:

```
(file, fwritec 'a' file)
```

- Function arguments with no uniqueness attributes added to the classical type are considered as "non-unique": there are no reference requirements for these arguments. The function is only allowed to have *read access* (as usual in a functional language) even if in some of the function applications the actual argument appears to have reference count 1.

```
freadc:: File -> (Char, File)
```

The function `freadc` can be applied to both a unique as well as non-unique file. This is fine since the function only wants read access on the file. The type indicates that the result is always a non-unique file. Such a file can be passed for further reading, but not for further writing.

To indicate that functions don't change uniqueness properties of arguments, one can use `attribute variables`.

The simplest example is the identity functions that can be typed as follows:

```
id:: u:a -> u:a
id x = x
```

Here `a` is an ordinary type variable, whereas `u` is an attribute variable. If `id` is applied to an unique object the result is also unique (in that case `u` is instantiated with the concrete attribute `*`). Of course, if id is applied to a non-unique object, the result remains non-unique. As with ordinary type variables, attribute variables should be instantiated uniformly.

A more interesting example is the function `freadc` that is typed as

```
freadc:: u:File -> u:(Char, u:File)
```

Again `freadc` can be applied to both unique and non-unique files. In the first case the resulting file is also unique and can, for example, be used for further reading or writing. Moreover, observe that not only the resulting file is attributed, but also the tuple containing that file and the character that has been read. This is due to the so called *uniqueness propagation rule*; see below.

To summarize, uniqueness typing makes it possible to update objects destructively within a purely functional language. For the development of real world applications (which manipulate files, windows, arrays, databases, states etc.) this is an indispensable property.

9.2 Attribute Propagation

Having explained the general ideas of uniqueness typing, we can now focus on some details of this typing system.

If a unique object is stored in a data structure, the data structure itself becomes unique as well. This *uniqueness propagation rule* prevents that unique objects are shared indirectly via the data structure in which these objects are stored. To explain this form of hidden sharing, consider the following definition of the function head

```
head:: [*a] -> *a
head [hd:tl] = hd
```

The pattern causes head to have access to the "deeper" arguments `hd` and `tl`. Note that head does not have any uniqueness requirements on its direct list argument. This means that in an application of head the list might be shared, as can be seen in the following function heads

```
heads list = (head list, head list)
```

If one wants to formulate uniqueness requirements on, for instance, the `hd` argument of `head`, it is *not* sufficient to attribute the corresponding type variable `a` with `*`; the surrounding list itself should also become unique. One can easily see that, without this additional requirement the `heads` example with type

```
heads:: [*a] -> (*a,*a)
heads list = (head list, head list)
```

would still be valid although it delivers the same object twice. By demanding that the surrounding list becomes unique as well, (so the type of `head` becomes `head:: *[*a] -> *a`) the function `heads` is rejected. In general one could say that uniqueness *propagates outwards*.

Some of the readers will have noticed that, by using attribute variables, one can assign a more general uniqueness type to head:

```
head:: u:[u:a] -> u:a
```

The above propagation rule imposes additional (implicit) restrictions on the attributes appearing in type specifications of functions.

Another explicit way of indicating restrictions on attributes is by using *coercion statements*. These statements consist of attribute variable inequalities of the form `u <= v`. The idea is that attribute substitutions are only allowed if the resulting attribute inequalities are valid, i.e. not resulting in an equality of the form

'non-unique ≤ unique'.

The use of coercion statements is illustrated by the next example in which the uniqueness type of the well-known append function is shown.

```
append:: v:[u:a] w:[u:a] -> x:[u:a],     [v<=u, w<=u, x<=u,w<=x]
```

The first three coercion statements express the uniqueness propagation for lists: if the elements `a` are unique (by choosing `*` for `u`) these statements force `v`,`w` and `x` to be instantiated with `*` also. (Note that `u <= * iff u = *`.) The statement `w <= x` expresses that the spine uniqueness of `append`'s result depends only on the spine attribute `w` of the second argument.

In CLEAN it is permitted to omit attribute variables and attribute inequalities that arise from propagation properties; these will be added automatically by the type system. As a consequence, the following type for `append` is also valid.

```
append:: [u:a] w:[u:a] -> x:[u:a],     [w<=x]
```

Of course, it is always allowed to specify a more specific type (by instantiating type or attribute variables). All types given below are valid types for `append`.

```
append:: [u:a] x:[u:a] -> x:[u:a],
append:: *[*Int] *[*Int] -> *[*Int],
append:: [a] *[a] -> *[a].
```

To make types more readable, CLEAN offers the possibility to use *anonymous* attribute variables. These can be used as a shorthand for indicating attribute variables of which the actual names are not essential. This allows us to specify the type for `append` as follows.

```
append:: [.a] w:[.a] -> x:[.a],    [w<=x]
```

The type system of CLEAN will substitute real attribute variables for the anonymous ones. Each dot gives rise to a new attribute variable except for the dots attached to type variables: type variables are attributed uniformly in the sense that all occurrences of the same type variable will obtain the same attribute. In the above example this means that all dots are replaced by one and the same new attribute variable.

9.3 Defining New Types with Uniqueness Attributes

```
AlgebraicTypeDef    =  ::TypeLhs  = ConstructorDef
                                    {| ConstructorDef} ;
ConstructorDef      =  [ExistQuantVariables] ConstructorName {ArgType} {& ClassConstraints}
                    |  [ExistQuantVariables] (ConstructorName) [FixPrec] {ArgType} {& ClassConstraints}

TypeLhs             =  [*] TypeConstructor {[*]TypeVariable}
TypeConstructor     =  TypeName

ExistQuantVariables =  E.{TypeVariable }+:
UnivQuantVariables  =  A.{TypeVariable }+:

BrackType           =  [Strict] [UnqTypeAttrib] SimpleType
ArgType             =  BrackType
                    |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type [ClassContext])
UnqTypeAttrib       =  *
                    |  UniqueTypeVariable:
                    |  .
```

As can be inferred from the syntax, the attributes that are actually allowed in data type definitions are '*' and '.'; attribute variables are not permitted. The (unique) * attribute can be used at any subtype whereas the (anonymous). attribute is restricted to non-variable positions.

If no uniqueness attributes are specified, this does not mean that one can only build non-unique instances of such a data type. Attributes not explicitly specified by the programmer are added automatically by the type system. To explain this standard uniqueness attribution mechanism, first remember that the types of data constructors are not specified by the programmer but derived from their corresponding data type definition.

For example, the (classical) definition of the `List` type

```
:: List a = Cons a (List a) | Nil
```

leads to the following types for its data constructors:

```
Cons:: a (List a) -> List a
Nil:: List a
```

To be able to create unique instances of data types, the standard attribution of CLEAN will automatically derive appropriate uniqueness variants for the types of the corresponding data constructors. Such a uniqueness variant is obtained via a consistent attribution of all types and subtypes appearing in a data type definition. Here, consistency means that such an attribution obeys the following rules (assume that we have a type definition for some type `T`).

1) Attributes that are explicitly specified are adopted.
2) Each (unattributed) type variable and each occurrence of `T` will receive an attribute variable. This is done in a uniform way: equal type variables will receive equal attributes, and all occurrence of `T` are also equally attributed.
3) Attribute variables are added at non-variable positions if they are required by the propagation properties of the corresponding type constructor. The attribute variable that is chosen depends on the argument types of this constructor: the attribution scheme takes the attribute variable of first argument appearing on a propagating position (see example below).
4) All occurrences of the `.`attribute are replaced by the attribute variable assigned to the occurrences of `T`.

Example of standard attribution for data constructors. For `Cons` the standard attribution leads to the type

```
Cons:: u:a v:(List u:a) -> v:List u:a, [v<=u]
```

The type of `Nil` becomes

```
Nil:: v:List u:a, [v<=u]
```

Consider the following `Tree` definition

```
:: Tree a  =  Node a [Tree a]
```

The type of the data constructor `Node` is

```
Node:: u:a v:[v:Tree u:a] -> v:Tree u:a, [v<=u]
```

Another `Tree` variant.

```
:: *Tree *a  =  Node a *[Tree a]
```

leading to

```
Node:: *a *[*Tree *a] -> *Tree *a
```

Note that, due to propagation, all subtypes have become unique.

Next, we will formalize the notion of uniqueness propagation. We say that an argument of a type constructor, say `T`, is propagating if the corresponding type variable appears on a propagating position in one of the types used in the right-hand side of `T`'s definition. A propagating position is characterized by the fact that it is not surrounded by an arrow type or by a type constructor with non-propagating arguments. Observe that the definition of propagation is cyclic: a general way to solve this problem is via a fixed-point construction.

Example of the propagation rule. Consider the (record) type definition for `Object`.

```
Object a b::  {state:: a, fun:: b -> a}
```

The argument `a` is propagating. Since `b` does not appear on a propagating position inside this definition, `Object` is not propagating in its second argument.

9.4 Uniqueness and Sharing

The type inference system of CLEAN will derive uniqueness information *after* the classical Milner/Mycroft types of functions have been inferred (see 4.3). As explained in Section 9.1, a function may require a *non-unique* object, a *unique* object or a *possibly unique* object. Uniqueness of the result of a function will depend on the attributes of its arguments and how the result is constructed. Until now, we distinguished objects with reference count 1 from objects with a larger reference count: only the former might be unique (depending on the uniqueness type of the object itself). In practice, however, one can be more liberal if one takes the evaluation order into account. The idea is that multiple reference to an (unique) object are harmless if one knows that only one of the references will be present at the moment it is accessed destructively. This has been used in the following function.

```
AppendAorB:: *File -> *File
AppendAorB file
    | isEven position = fwritec 'a' file
                      = fwritec 'b' file
where
    position = sfposition file
```

When the right-hand side of `AppendAorB` is evaluated, the guard is determined first (so access from `sfposition` to `file` is not unique), and subsequently one of the alternatives is chosen and evaluated. Depending on `position`, either the reference from the first `fwritec` application to `file` or that of the second application is left and therefore unique.

For this reason, the uniqueness type system uses a kind of *sharing analysis*. This sharing analysis is input for the uniqueness type system itself to check uniqueness type consistency (see 9.3). The analysis will label each *reference* in the right-hand side of a function definition as *read-only* (if destructive access might be dangerous) or *write-permitted* (otherwise). Objects accessed via a read-only reference are always non-unique. On the other hand, uniqueness of objects accessed via a reference labeled with *write-permitted* solely depends on the types of the objects themselves.

Before describing the labeling mechanism of CLEAN we mention that the "lifetime" of references is determined on a syntactical basis. For this reason we classify references to the same expression in a function definition (say for `f`) according to their estimated run-time use, as *alternative*, *observing* and *parallel*.

- Two references are *alternative* if they belong to different alternatives of `f`. Note that alternatives are distinguished by patterns (including `case` expressions) or by guards.
- A reference `r` is *observing* w.r.t. a reference `r'` if the expression containing `r'` is either (1) guarded by an expression or (2) preceded by a strict let before expression containing `r`.
- Otherwise, references are in *parallel*.

The rules used by the sharing analysis to label each reference are the following.

- A reference, say `r`, to a certain object is labeled with read-only if there exist another reference, say `r'`, to the same object such that either `r` is observing w.r.t `r'` or `r` and `r'` are in parallel.
- Multiple references to *cyclic structures* are always labeled as read-only.
- All other references are labeled with write-permitted.

Unfortunately, there is still a subtlety that has to be dealt with. Observing references belonging in a strict context do not always vanish totally after the expression containing the reference has been evaluated: further analysis appears to be necessary to ensure their disappearance. More concretely, suppose `e[r]` denotes the expression containing `r`. If the type of `e[r]` is a basic type then, after evaluation, `e[r]` will be reference-free. In particular, it does not contain the reference `r` anymore. However, If the type of `e[r]` is not a basic type it is assumed that, after evaluation, `e[r]` might still refer to r. But even in the latter case a further refinement is possible. The idea is, depending on `e[r]`, to correct the type of the object to which `r` refers partially in such way that only the parts of this object that are still shared lose their uniqueness.

Consider, for example, the following rule

```
f l =
#! x = hd (hd l)
= (x, l)
```

Clearly, x and l share a common substructure; `x` is even part of `l`. But the whole "spine" of `l` (of type `[[...]]`) does not contain any new external references. Thus, if `l` was spine-unique originally, it remains spine unique in the result of `f`. Apparently, the access to `l` only affected part of `l`'s structure. More technically, the type of `l` itself is corrected to take the partial access on `l` into account. In the previous example, `x`, regarded as a function on `l` has type `[[a]] -> a`. In `f`'s definition the part of `l`'s type corresponding to the variable a is made non-unique. This is clearly reflected in the derived type for `f`, being

```
f:: u:[w:[a]] -> (a,v:[x:[a]]), [w <= x, u <= v]
```

In CLEAN this principle has been generalized: If the strict let expression `e[r]` regarded as a function on r has type
`T (... a...) -> a`

Then the a-part of the type of the object to which r refers becomes non-unique; the rest of the type remains unaffected. If the type of `e[r]` is not of the indicated form, `r` is not considered as an observing reference (w.r.t. some reference `r'`), but, instead, as in parallel with `r'`.

9.4.1 Higher Order Uniqueness Typing

Higher-order functions give rise to partial (often called *curried*) applications, i.e. applications in which the actual number of arguments is less than the arity of the corresponding symbol. If these partial applications contain unique sub-expressions one has to be careful.

Consider, for example the following the function `fwritec` with type

```
fwritec:: *File Char -> *File
```

in the application

```
fwritec unifile
```

(assuming that unifile returns a unique file). Clearly, the type of this application is of the form `o:(Char -> *File)`. The question is: what kind of attribute is `o`? Is it a variable, is it *, or, is it not unique? Before making a decision, one should notice that it is dangerous to allow the above application to be shared. For example, if the expression `fwritec unifile` is passed to a function

```
WriteAB write_fun = (write_fun 'a', write_fun 'b')
```

Then the argument of `fwritec` is no longer unique at the moment one of the two write operations take place. Apparently, the `fwritec unifile` expression is *essentially* unique: its reference count should never become greater than 1. To prevent such an essentially unique expression from being copied, CLEAN considers the `->` type constructor in combination with the `*` attribute as special: it is not permitted to discard its uniqueness. Now, the question about the attribute `o` can be answered: it is set to `*`. If `WriteAB` is typed as follows

```
WriteAB:: (Char -> u:File) -> (u:File, u:File)
WriteAB write_fun = (write_fun 'a', write_fun 'b')
```

the expression `WriteAB (fwritec unifile)` is rejected by the type system because it does not allow the argument of type `*(Char -> *File)` to be coerced to `(Char -> u:File)`. One can easily see that it is impossible to type `WriteAB` in such a way that the expression becomes typable.

To define data structures containing Curried applications it is often convenient to use the (anonymous) `.` attribute. Example

```
:: Object a b = { state:: a, fun::.(b -> a) }
new:: * Object *File Char
new = { state = unifile, fun = fwritec unifile }
```

By adding an attribute variable to the function type in the definition of `Object`, it is possible to store unique functions in this data structure. This is shown by the function `new` . Since `new` contains an essentially unique expression it becomes essentially unique itself. So, `new` can never lose its uniqueness, and hence, it can only be used in a context in which a unique object is demanded.

Determining the type of a curried application of a function (or data constructor) is somewhat more involved if the type of that function contains attribute variables instead of concrete attributes. Mostly, these variables will result in additional coercion statements. as can be seen in the example below.

```
Prepend:: u:[.a] [.a] -> v:[.a],  [u<=v]
Prepend a b = Append b a

PrependList:: u:[.a] -> w:([.a] -> v:[.a]),     [u<=v, w<=u]
PrependList a = Prepend a
```

Some explanation is in place. The expression `(PrependList some_list)` yields a function that, when applied to another list, say `other_list`, delivers a new list extended consisting of the concatenation of `other_list` and `some_list`. Let's call this final result new_list. If `new_list` should be unique (i.e. v becomes `*`) then, because of the coercion statement `u<=v` the attribute u also becomes `*`. But, if `u = *` then also `w = *`, for, `w<=u`. This implies that (arrow) type of the original expression `(PrependList some_list)` becomes unique, and hence this expression cannot be shared.

9.4.2 Uniqueness Type Coercions

As said before, offering a unique object to a function that requires a non-unique argument is safe (unless we are dealing with unique arrow types; see above). The technical tool to express this is via a coercion (subtype) relation based on the ordering

'unique' ≤ 'non-unique'

on attributes. Roughly, the validity of σ ≤ σ' depends subtype-wise on the validity of `u ≤ u'` with `u,u'` attributes in σ,σ'. One has, for example

`u:[v:[w:Int]]` ≤ `u':[v':[w':Int]]` iff `u ≤ u'`, `v ≤ v'`, `w ≤ w'`.

However, a few refinements are necessary. Firstly, the uniqueness constraints expressed in terms of coercion statements (on attribute variables) have to be taken into account. Secondly, the coercion restriction on arrow types should be handled correctly. And thirdly, due to the so-called *contravariance* of `->` in its first argument we have that

`u`:(σ -> σ') ≤ `u`:(τ -> τ') iff τ ≤ σ, σ' ≤ τ'

Since `->` may appear in the definitions of algebraic type constructors, these constructors may inherit the co- and contravariant subtyping behavior with respect to their arguments. We can classify the 'sign' of the arguments of each type constructor as + (positive, covariant), - (negative, contravariant) or top (both positive and negative). In general this is done by analysing the (possible mutually recursive) algebraic type definitions by a fixed-point construction, with basis sign(`->`) = (−,+).

Example: `a` has sign `T`, `b` has sign + in

```
::FunList a b = FunCons (a, a -> b) (FunList a b)
              | FunNil
```

This leads to the following coercion rules

5) Attributes of two corresponding type variables as well as of two corresponding arrow types must be equal.
6) The sign classification of each type constructor is obeyed. If, for instance, the sign of T's argument is negative, then T σ ≤ T σ' iff σ' ≤ σ
7) In all other cases, the validity of a coercion relation depends on the validity of `u ≤ u'`, where `u,u'` are attributes of the two corresponding subtypes.

The presence of sharing inherently causes a (possibly unique) object to become non-unique, if it is accessed via a read-only reference. In CLEAN this is achieved by a type correction operation that converts each unique type S to its smallest non-unique supertype, simply by making the outermost attribute of S non-unique. Note that this operation fails if S is a function type.

9.5 Combining Uniqueness Typing and Overloading

An overloaded function actually stands for a collection of real functions. The types of these real functions are obtained from the type of the overloaded function by substituting the corresponding instance type for the class variable. These instance types may contain uniqueness information, and, due to the propagation requirement, the above-mentioned substitution might give rise to uniqueness attributes overloaded type specification.

Consider, for instance, the identity class

```
class id a:: a -> a
```

If we want to define an instance of `id` for lists, say `id L`, which leaves uniqueness of the list elements intact, the (fully expanded) type of `idL` becomes

```
instance id L v:[u:a] -> v:[u:a]
```

However, as said before, the type specification of such an instance is not specified completely: it is derived from the overloaded type in combination with the instance type (i.e. `[...]` in this particular example).

In CLEAN we require that the type specification of an overloaded operator anticipates on attributes arising from uniqueness propagation, that is, the uniqueness attribute of the class variable should be chosen in such a way that for any instance type this 'class attribute' does not conflict with the corresponding uniqueness attribute(s) in the fully expanded type of this instance. In the above example this means that the type of `id` becomes

```
class id a:: a -> a
```

Another possibility is

```
class id a:: *a -> *a
```

However, the latter version of `id` will be more restrictive in its use, since it will always require that its argument is unique.

9.5.1 Constructor Classes

The combination of uniqueness typing and constructor classes (with their higher-order class variables) introduces another difficulty. Consider, for example, the overloaded map function.

```
class map m:: (a -> b) (m a) -> m b
```

Suppose we would add (distinct) attribute variables to the type variables `a` and `b` (to allow 'unique instances' of map)

```
class map m:: (.a ->.b) (m .a) -> m .b
```

The question that arises is: Which attributes should be added to the two applications of the class variable `m`? Clearly, this depends on the actual instance type filled in for `m`. E.g., if `m` is instantiated with a propagating type constructor (like `[]`), the attributes of the applications of `m` are either attribute variables or the concrete attribute 'unique'. Otherwise, one can choose anything.

Example

```
instance map []
where
    map f l = [ f x // x <- l ]

::  T a = C (Int -> a)

instance map T
where
    map f (C g) = C (f o g)
```

In this example, the respective expanded types of the instances are

```
map:: (u:a -> v:b) w:[u:a] -> x:[v:b], [w <= u, x <= v]
map:: (u:a -> v:b) (T u:a) -> T v:b
```

The type system of CLEAN requires that a possible propagation attribute is explicitly indicated in the type specification of the overloaded function. In order to obtain versions of `map` producing spine unique data structures, its overloaded type should be specified as follows:

```
class map m:: (.a ->.b).(m. a) ->.(m. b)
```

This type will provide that for an application like

```
map inc [1,2,3]
```

indeed yields a spine unique list.

Observe that if you would omit the (anonymous) attribute variable of the second argument, the input data structure cannot contain unique data on propagating positions, e.g. one could not use such a version of `map` for mapping a destructive write operator on a list of unique files.

In fact, the propagation rule is used to translate uniqueness properties of objects into uniqueness properties of the data structures in which these objects are stored. As said before, in some cases the actual data structures are unknown.

Consider the following function

```
DoubleMap f l = (map f l, map f l)
```

The type of this function is something like

Clearly, `l` is duplicated. However, this does not necessarily mean that a cannot be unique anymore. If, for instance, `m` is instantiated with a non-propagating type constructor (like T as defined on the previous page) then uniqueness of a is still permitted. On the other hand, if `m` is instantiated with a propagating type constructor, a unique instantiation of a should be disapproved. In CLEAN, the type system 'remembers' sharing of objects (like `l` in the above example) by making the corresponding type attribute non-unique. Thus, the given type for `DoubleMap` is exactly the type inferred by CLEAN's type system. If one tries to instantiate `m` with a propagating type constructor, and, at the same type, a with some unique type, this will fail.

The presence of higher-order class variables, not only influences propagation properties of types, but also the coercion relation between types. These type coercions depend on the sign classification of type constructors. The problem with higher-order polymorphism is that in some cases the actual type constructors substituted for the higher order type variables are unknown, and therefore one cannot decide whether coercions in which higher-order type variable are involved, are valid.

Consider the functions

```
double x = (x,x)
dm f l = double (map f l)
```

Here, map's result (of type .(m .a)) is coerced to the non-unique supertype (m .a). However, this is only allowed if m is instantiated with type constructors that have no coercion restrictions. E.g., if one tries to substitute *WriteFun for m, where

```
WriteFun a = C.(a -> *File)
```

this should fail, for, *WriteFun is *essentially* unique. The to solve this problem is to restrict coercion properties of type variable applications (m σ) to

u:(m σ) = u:(m τ) iff σ ≤ τ && τ ≤ σ

A slightly modified version of this solution has been adopted in CLEAN. For convenience, we have added the following refinement. The instances of type constructors classes are restricted to type constructors with no coercion restrictions. Moreover, it is assumed that these type constructors are uniqueness propagating. This means that the WriteFun cannot be used as an instance for map. Consequently, our coercion relation we can be more liberal if it involves such class variable applications.

Overruling this requirement can be done adding the anonymous attribute. the class variable. E.g.

```
class map.m:: (.a ->.b).(m. a) ->.(m. b)
```

Now

```
instance map WriteFun
where
    map...
```

is valid, but the coercions in which (parts of) map's type are involved are now restricted as explained above. To see the difference between the two indicated variants of constructor variables, we slightly modify map's type.

To see the difference between the two indicated variants of constructor variables, we slightly modify map's type.

```
class map m:: (.a ->.b) *(m. a) ->.(m. b)
```

Without overruling the instance requirement for m the type of dm (dm as given on the previous page) becomes.

```
dm:: (.a ->.b) *(m.a) ->.(m b, m b)
```

Observe that the attribute of disappeared due to the fact that each type constructor substituted for m is assumed to be propagating.

If one explicitly indicates that there are no instance restriction for the class variable m (by attributing m with.), the function dm becomes untypable.

9.6 Higher-Order Type Definitions

We will describe the effect of uniqueness typing on type definitions containing higher-order type variables. At it turns out, this combination introduces a number of difficulties which would make a full description very complex. But even after skipping a lot of details we have to warn the reader that some of the remaining parts are still hard to understand.

As mentioned earlier, two properties of newly defined type constructor concerning uniqueness typing are important, namely, propagation and sign classification. One can probably image that, when dealing with higher-order types the determination on these properties becomes more involved. Consider, for example, the following type definition.

```
::  T m a = C (m a)
```

The question whether T is propagating in its second argument cannot be decided by examining this definition only; it depends on the actual instantiation of the (higher-order) type variable m. If m is instantiated with a propagating type constructor, like [], then T becomes propagating in its second argument as well. Actually, propagation is not only a property of type constructors, but also of types themselves, particularly of 'partial types' For example, the partial type [] is propagating in its (only) argument (Note that the number of arguments a partial type expects, directly follows from the kinds of the type constructors that have been used). The type T [] is also propagating in its argument, so is the type T ((,) Int)).

The analysis in CLEAN that determines propagation properties of (partial) types has been split into two phases. During the first phase, new type definitions are examined in order to determine the propagation dependencies between the arguments of each new type constructor. To explain the idea, we return to our previous example.

```
::  T m a = C (m a)
```

First observe that the propagation of the type variable `m` is not interesting because `m` does not stand for 'real data' (which is always of kind `*`). We associate the propagation of `m` in `T` with the position(s) of the occurrence(s) of `m`'s applications. So in general, `T` is propagating in a higher-order variable `m` if one of `m`'s applications appears on a propagating position in the definition of T. Moreover, for each higher order type variable, we determine the propagation properties of all first order type variables in the following way: `m` is propagating in `a`, where `m` and `a` are higher-order respectively first-order type variables of `T`, if `a` appears on a propagating position in one of `m`'s applications. In the above example, `m` is propagating in `a`, since `a` is on a propagating position in the application `(m a)`. During the second phase, the propagation properties of (partial) types are determined using the results of the first phase. This (roughly) proceeds as follows. Consider the type `T` σ for some (partial) type σ, and `T` as defined earlier. First, determine (recursively) the propagation of σ. Then the type `T` σ is propagating if (1) σ is propagating, (2) `T` is propagating in `m`, and moreover (3) `m` is propagating in `a` (the second argument of the type constructor). With `T` as defined above, (2) and (3) are fulfilled. Thus, for example `T []` is propagating and therefore also `T (T [])`. Now define

```
::  T2 a = C2 (a -> Int)
```

Then `T T2` is not propagating.

The adjusted uniqueness propagation rule (see also...) becomes:

- Let σ,τ be two uniqueness types. Suppose σ has attribute `u`. Then, if τ is propagating the application (τ σ) should have an attribute `v` such that `v` ≤ `u`.

Some of the readers might have inferred that this propagation rule is a 'higher-order' generalization of the old 'first-order' propagation rule.

As to the sign classification, we restrict ourselves to the remark that that sign analysis used in CLEAN is adjusted in a similar way as described above for the propagation case.

Example

```
::  T m a = C ((m a) -> Int)
```

The sign classification of `T` if (-,⊥). Here ⊥ denotes the fact the `a` is neither directly used on a positive nor on a negative position. The sign classification of m w.r.t. `a` is +. The partial type `T []` has sign -, which e.g. implies that
`T [] Int ≤ T [] *Int`

The type `T T2` (with `T2` as defined on the previous page) has sign +, so
`T T2 Int ≥ T T2 *Int`

It will be clear that combining uniqueness typing with higher-order types is far from trivial: the description given above is complex and moreover incomplete. However explaining all the details of this combination is far beyond the scope of the reference manual.

9.7 Destructive Updates using Uniqueness Typing

So, it is *allowed* to update a uniquely typed function argument (`*`) destructively when the argument does not reappear in the function result. The question is: when does the compiler indeed make use of this possibility.

Destructive updates takes place in some predefined functions and operators which work on predefined data structures such arrays (&-operator) and files (writing to a file). Arrays and files are intended to be updated destructively and their use can have a big influence on the space and time behavior of your application (a new node does not have to be claimed and filled, the garbage collector is invoked less often and the locality of memory references is increased).

Performing destructive updates is only sensible when information is stored in nodes. Arguments of basic type (`Int`, `Real`, `Char` or `Bool`) are stored on the B-stack or in registers and it therefore does not make sense to make them unique.

The CLEAN compiler also has an option to re-use user-defined unique data structures: the space being occupied by a function argument of unique type will under certain conditions be reused destructively to construct the function result when (part of) this result is of the same type. So, a more space and time efficient program can be obtained by turning heavily used data structures into unique data structures. This is not just a matter of changing the uniqueness type attributes (like turning a lazy data structure into a strict one). A unique data structure also has to be used in a 'single threaded' way (see Chapter 4). This means that one might have to restructure parts of the program to maintain the unicity of objects.

The compiler will do compile-time garbage collection for user defined unique data-structures only in certain cases. In that case run-time garbage collection time is reduced. It might even drop to zero. It is also possible that you gain even more than just garbage collection time due to better cache behavior.

Chapter 10

Strictness, Macros and Efficiency

Programming in a functional language means that one should focus on algorithms and without worrying about all kinds of efficiency details. However, when large applications are being written it may happen that this attitude results in a program that is unacceptably inefficient in time and/or space.

In this Chapter we explain several kinds of annotations and directives that can be defined in CLEAN. These annotations and directives are designed to give the programmer some means to influence the time and space behavior of CLEAN applications.

CLEAN is by default a *lazy* language: applications will only be evaluated when their results are needed for the final outcome of the program. However, lazy evaluation is in general not very efficient. It is much more efficient to compute function arguments in advance (*strict* evaluation) when it is known that the arguments will be used in the function body. By using strictness annotations in type definitions the evaluation order of data structures and functions can be changed from lazy to strict. This is explained in Section 10.1.

One can define constant graphs on the global level also known as `Constant Applicative Forms` (see Section 10.2). Unlike constant functions, these constant graphs are shared such that they are computed only one. This generally reduces execution time possibly at the cost of some heap space needed to remember the shared graph constants.

Macro's (Section 10.3) are special functions that will already be substituted (evaluated) at *compile-time*. This generally reduces execution time (the work has already been done by the compiler) but it will lead to an increase of object code.

10.1 Annotations to Change Lazy Evaluation into Strict Evaluation

CLEAN uses by default a *lazy evaluation strategy*: a redex is only evaluated when it is needed to compute the final result. Some functional languages (e.g. ML, Harper *et al.*) use an *eager* (*strict*) evaluation strategy and always evaluate all function arguments in advance.

10.1.1 Advantages and Disadvantages of Lazy versus Strict Evaluation

Lazy evaluation has the following advantages (+) / disadvantages (-) over eager (strict) evaluation:

- only those computations which contribute to the final result are computed (for some algorithms this is a clear advantage while it generally gives a greater expressive freedom);
- one can work with infinite data structures (e.g. [1..])
- it is unknown when a lazy expression will be computed (disadvantage for debugging, for controlling evaluation order);
- strict evaluation is in general much more efficient, in particular for objects of basic types, non-recursive types and tuples and records which are composed of such types;

+/- in general a strict expression (e.g. `2 + 3 + 4`) takes less space than a lazy one, however, sometimes the other way around (e.g. `[1..1000]`);

10.1.2 Strict and Lazy Context

Each expression in a function definition is considered to be either strict (appearing in a *strict context*: it has to be evaluated to strong root normal form) or lazy (appearing in a *lazy context* : not yet to be evaluated to strong root normal form) The following rules specify whether or not a particular expression is lazy or strict:

- a non-variable pattern is strict;
- an expression in a guard is strict;
- the expressions specified in a strict let-before expression are strict;
- the **root expression** is strict;
- the arguments of a function or data constructor in a strict context are strict when these arguments are being annotated as strict in the type definition of that function (manually or automatically) or in the type definition of the data constructor;
- all the other expressions are lazy.

Evaluation of a function will happen in the following order: patterns, guard, expressions in a strict let before expression, root expression (see also 3.1).

10.1.3 Space Consumption in Strict and Lazy Context

The space occupied by CLEAN structures depend on the kind of structures one is using, but also depends on whether these data structures appear in a strict or in a lazy context. To understand this one has to have some knowledge about the basic implementation of CLEAN (see Plasmeijer and Van Eekelen, 1993).

Graphs (see Chapter 1) are stored in a piece of memory called the heap. The amount of heap space needed highly depends on the kind of data structures that are in use. Graph structures that are created in a lazy context can occupy more space than graphs created in a strict context. The garbage collector in the run-time system of CLEAN automatically collects graphs that are not being used. The arguments of functions being evaluated are stored on a stack. There are two stacks: the A-stack, which contains references to graph nodes stored in the heap and the BC-stack which contains arguments of basic type and return addresses. Data structures in a *lazy context* are passed via references on the A-stack. Data structures of the *basic types* (`Int`, `Real`, `Char` or `Bool`) in a *strict context* are stored on the B-stack or in registers. This is also the case for these strict basic types when they are part of a *record* or *tuple* in a strict context.

Data structures living on the B-stack are passed *unboxed*. They consume less space (because they are not part of a node) and can be treated much more efficiently. When a function is called in a lazy context its data structures are passed in a graph node (*boxed*). The amount of space occupied is also depending on the arity of the function.

In the table below the amount of space consumed in the different situations is summarised (for the lazy as well as for the strict context). For the size of the elements one can take the size consumed in a strict context.

Type	*Arity*	*Lazy context (bytes)*	*Strict context (bytes)*	*Comment*
`Int,Bool`	–	8	4	
`Int (0≤n≤32), Char`	–	–	4	node is shared
`Real`	–	12	8	
`Small Record`	n	4 + Σ size elements	Σ size elements	total length≤12
`Large Record`	n	8 + Σ size elements	Σ size elements	
`Tuple`	2	12	Σ size elements	
	>2	8 + 4*n	Σ size elements	
`{a}`	n	20 + 4*n	12 + 4*n	
`!Int`	n	20 + 4*n	12 + 4*n	
`!Bool,!Char`	n	20 + 4*ceil(n/4)	12 + 4*ceil(n/4)	
`!Real`	n	20 + 8*n	12 + 8*n	
`!Tuple, !Record`	n	20 + size rec/tup*n	12 + size rec/tup*n	
`Hnf`	0	–	4 + size node	node is shared
	1	8	4 + size node	
	2	12	4 + size node	also for `[a]`
	>2	8 + 4*n	4 + size node	
`Pointer to node`	–	4	4	
`Function`	0,1,2	12	–	
	>2	4 + 4*n	–	

10.1.4 Time Consumption in Strict and Lazy Context

Strict arguments of functions can sometimes be handled much more efficiently than lazy arguments, in particular when the arguments are of basic type.

Example: functions with strict arguments of basic type are more efficient.

```
Ackerman:: !Int !Int -> Int
Ackerman 0 j = j+1
Ackerman i 0 = Ackerman (i-1) 1
Ackerman i j = Ackerman (i-1) (Ackerman i (j-1))
```

The computation of a lazy version of `Ackerman 3 7` takes 14.8 seconds + 0.1 seconds for garbage collection on an old fashion MacII (5Mb heap). When both arguments are annotated as strict (which in this case will be done automatically by the compiler) the computation will only take 1.5 seconds + 0.0 seconds garbage collection. The gain is one order of magnitude. Instead of rewriting graphs the calculation is performed using stacks and registers where possible. The speed is comparable with a recursive call in highly optimised C or with the speed obtainable when the function was programmed directly in assembly.

10.1.5 Changing Lazy into Strict Evaluation

So, lazy evaluation gives a notational freedom (no worrying about what is computed when) but it might cost space as well as time. In CLEAN the default lazy evaluation can therefore be turned into eager evaluation by adding strictness annotations to types.

Strict = `!`

This can be done in several ways:

- The CLEAN compiler has a built-in strictness analyzer based on *abstract reduction* Nöcker, 1993) (it can be optionally turned off). The analyzer searches for strict arguments of a function and annotate them internally as strict (see 10.1.1). In this way lazy arguments are *automatically* turned into strict ones. This optimization does not influence the termination behavior of the program. It appears that the analyzer can find much information. The analysis itself is quite fast.
- The strictness analyzer cannot find all strict arguments. Therefore one can also *manually* annotate a function as being strict in a certain argument or in its result (see 10.1.1).
- By using strictness annotations, a programmer can define (partially) strict data structures Nöcker and Smetsers, 1993; see 10.1.3). Whenever such a data structure occurs in a strict context (see 10.1.1), its strict components will be evaluated.
- The order of evaluation of expressions in a function body can also be changed from lazy to strict by using a strict let-before expression see 3.5.4).

One has to be careful though. When a programmer manually changes lazy evaluation into strict evaluation, the termination behavior of the program might change. It is only safe to put strictness annotations in the case that the function or data constructor is known to be strict in the corresponding argument which means that the evaluation of that argument in advance does not change the termination behavior of the program. The compiler is not able to check this.

10.2 Defining Graphs on the Global Level

Constant graphs can also be defined on a global level (for local constant graphs see 3.6).

GraphDef = Selector `=` [`:`] GraphExpr `;`

A *global graph definition* defines a global constant (closed) graph, i.e. a graph which has the same scope as a global function definition (see 2.1). The selector variables that occur in the selectors of a global graph definition have a global scope just as globally defined functions.

Special about *global* graphs (in contrast with *local* graphs) is that they are *not* garbage collected during the evaluation of the program A global graph can be compared with a *CAF* (*Constant Applicative Form*): its value is computed at most once and remembered at run-time. A global graph can save execution-time at the cost of permanent space consumption.

Syntactically the definition of a graph is distinguished from the definition of a function by the symbol which separates the left-hand side from the right-hand side: "=" or "=>" is used for functions, while "=" is used for local graphs and "=:" for global graphs. However, in general "=" is used both for functions and local graphs. Generally it is clear from the context which is meant (functions have parameters, selectors are also easy recognisible). However, when a simple constant is defined the syntax is ambiguous (it can be a constant function definition as well as a constant graph definition).

To allow the use of the "=" whenever possible, the following rule is followed. Locally constant definitions are *by default* taken to be *graph* definitions and therefore shared, globally they are *by default* taken to be *function* definitions (see 3.1) and therefore recomputed. If one wants to obtain a different behavior one has to explicit state the nature of the constant definition (has it to be shared or has it to be recomputed) by using "=:" (on the global level, meaning it is a constant graph which is shared) or "=>" (on the local level, meaning it is a constant function and has to be recomputed).

Global constant graph versus global constant function definition: `biglist1` is a *graph* which is computed only once, `biglist3` and `biglist2` is a constant *function* which is computed every time it is applied.

```
biglist1 =   [1..10000]              // a constant function (if defined globally)
biglist2 =:  [1..10000]              // a graph (if defined globally)
biglist3 =>  [1..10000]              // a constant function
```

A graph saves execution-time at the cost of space consumption. A constant function saves space at the cost of execution time. So, use graphs when the computation is time-consuming while the space consumption is small and constant functions in the other case.

10.3 Defining Macros

Macros are functions (rewrite rules) which are applied at *compile-time* instead of at *run-time*. Macro's can be used to define constants, create in-line substitutions, rename functions, do conditional compilation etc. With a macro definition one can, for instance, assign a name to a constant such that it can be used as pattern on the left-hand side of a function definition.

At compile-time the right-hand side of the *macro definition* will be substituted for every application of the macro in the scope of the macro definition. This saves a function call and makes basic blocks larger (see Plasmeijer and Van Eekelen, 1993) such that *better* code can be generated. A disadvantage is that also *more* code will be generated. Inline substitution is also one of the regular optimisations performed by the CLEAN compiler. To avoid code explosion a compiler will generally not substitute big functions. Macros give the programmer a possibility to control the substitution process manually to get an optimal trade-off between the efficiency of code and the size of the code.

```
MacroDef          =  [MacroFixityDef]
                     DefOfMacro
MacroFixityDef    =  (FunctionName) [FixPrec] ;
DefOfMacro        =  Function {Variable} :== FunctionBody ;
                     [LocalFunctionAltDefs]
```

The compile-time substitution process is guaranteed to terminate. To ensure this some restrictions are imposed on Macro's (compared to common functions). Only variables are allowed as formal argument. A macro rule always consists of a single alternative. Furthermore,

8) Macro definitions are not allowed to be cyclic to ensure that the substitution process terminates.

Example of a macro definition.

```
Black    :== 1                                  // Macro definition
White    :== 0                                  // Macro definition

:: Color :== Int                                // Type synonym definition

Invert:: Color -> Color                         // Function definition
Invert Black = White
Invert White = Black
```

Example: macro to write (a?b) for lists instead of [a:b] and its use in the function map.

```
(?) infixr 5                                        // Fixity of Macro
(?) h t :== [h:t]                                   // Macro definition of operator

map:: (a -> b) [a] -> [b]
map f (x?xs) = f x ? map f xs
map f []     = []
```

Notice that macros can contain local function definitions. These local definitions (which can be recursive) will also be substituted inline. In this way complicated substitutions can be achieved resulting in efficient code.

Example: macros can be used to speed up frequently used functions. See for instance the definition of the function `foldl` in `StdList`.

```
foldl op r l :== foldl r l                          // Macro definition
where
    foldl r []    = r
    foldl r [a:x] = foldl (op r a) x

sum list = foldl (+) 0 list
```

After substitution of the macro `foldl` a very efficient function `sum` will be generated by the compiler:

```
sum list = foldl 0 list
where
    foldl r []    = r
    foldl r [a:x] = foldl ((+) r a) x
```

The expansion of the macros takes place before type checking. Type specifications of macro rules are not possible. When operators are defined as macros, fixity and associativity can be defined.

10.4 Efficiency Tips

Here are some additional suggestions how to make your program more efficient:

- Use the CLEAN profiler to find out which frequently called functions are consuming a lot of space and/or time. If you modify your program, these functions are the ones to have a good look at.
- Transform a recursive function to a tail-recursive function.
- It is better to accumulate results in parameters instead of in the right-hand side results.
- It is better to use records instead of tuples.
- Arrays can be more efficient than lists since they allow constant access time on their elements and can be destructive updated.
- When functions return multiple ad-hoc results in a tuple put these results in a strict tuple instead (can be indicated in the type).
- Use strict data structures whenever possible.
- Export the strictness information to other modules (the compiler will warn you if you don't).
- Make function strict in its arguments whenever possible.
- Use macros for simple constant expressions or frequently used functions.
- Use CAF's and local graphs to avoid recalculation of expressions.
- Selections in a lazy context can better be transformed to functions which do a pattern match.
- Higher order functions are nice but inefficient (the compiler will try to convert higher order function into first order functions).
- Constructors of high arity are inefficient.
- Increase the heap space in the case that the garbage collector takes place too often.

Chapter 11

Foreign Language Interface

The tool htoclean can be used to generate interfaces to C functions. This is not discussed in this manual.

How to call Clean functions from C is discussed in section 11.1.

ABC instructions of the virtual machine for Clean can be used. This is explained in section 11.2.

11.1 Foreign Export

Some Clean functions can be called from C using foreign export. This is possible if:

- The function is exported.
- All arguments are annotated as being strict (see 3.7.5).
- The arguments and result type is either of the following:
  - Int
  - Real
  - {#Char}
  - {#Int}
  - {#Real}
  - A tuple of these strictly annotated types (including tuples).

The following syntax is used in an implementation module to export a function to a foreign language:

```
ForeignExportDef        =  foreign export [ ccall | stdcall ] FunctionName ;
```

The calling convention may be specified by prefixing the function name with ccall or stdcall. The default is ccall.

To pass an argument from C of type:

- Int, use a C integer type that has the same size as an Int in Clean. On 32 bit platforms this is 4 bytes, on 64 bit platforms 8 bytes. Usually long has the right size, except on 64 bit Windows __int64 can be used instead.
- Real, use type double.
- {#Char}, pass the address of the string. The first 4 (on 32 bit platforms) or 8 (on 64 bit platforms) bytes should contain the number of characters. The characters of the string are stored in the following bytes. The string is copied to the heap of Clean, and this copy is used in Clean.
- {#Int}, pass the address of the array. The elements of the array have the same size as an Int in Clean. The number of elements should be stored in 4 bytes at offset -8 (32 bit) or 8 bytes at offset -16 (64 bit). The array is copied to the heap of Clean, and this copy is used in Clean.
- {#Real}, pass the address of the array. The elements of the array have the same size as a Real in Clean (8 bytes) and a double in C. The number of elements should be stored in the same way as for {#Int}. The array is copied to the heap of Clean, and this copy is used in Clean.
- Tuple. The elements are passed as separate arguments as described above, in the same order as in the tuple.

Preferably, the macros in the file Clean.h (part of the tool htoclean) should be used to access strings and arrays.

If result of the function is not a tuple, the result is passed in the same way as an argument, except that strings and arrays are not copied. The address of the string or array in the heap of Clean is passed to C. This string or array may only be used until the next call or return to Clean, because the Clean runtime system may deallocate or move the array.

If multiple values are yielded, because the result is a tuple, the result type in C is void. To return the values, for each value an additional argument with the address where the result should be stored is added (at the end, in the same order as in the tuple). For example, if the result has type (Real, Int), an additional double * and long * (__int64 * for 64 bit Windows) is added.

11.2 Using ABC instructions

Function can be implemented using ABC instructions from the virtual machine used by Clean:

```
ABCCodeFunctionDef      =   Function {Pattern} = code [inline] { ABCInstructions }
```

By adding `inline` the ABC instructions will be inlined if the function is called in a strict context.

This is used to define primitive functions of the StdEnv, for example integer addition. htoclean generates calls to C functions using the ABC instruction ccall.

Appendix A

Context-Free Syntax Description

In this appendix the context-free syntax of CLEAN is given. Notice that the layout rule (see 2.3.3) permits the omission of the semi-colon (';') which ends a definition and of the braces ('{' and '}') which are used to group a list of definitions.

The following notational conventions are used in the context-free syntax descriptions:

[notion]	means that the presence of notion is optional
{notion}	means that notion can occur zero or more times
{notion}+	means that notion occurs at least once
{notion}-*list*	means one or more occurrences of notion separated by commas
terminals	are printed in **9 pts courier bold brown**
keywords	are printed in **9 pts courier bold red**
terminals	that can be left out in layout mode are printed in **9 pts courier bold blue**
{notion}/ str	means the longest expression not containing the string str

A.1 Clean Program

```
CleanProgram          =  {Module}+
Module                =  DefinitionModule
                      |  ImplementationModule
DefinitionModule      =  definition module ModuleName ;
                         {DefDefinition}
                      |  system module ModuleName ;
                         {DefDefinition}
ImplementationModule  =  [implementation] module ModuleName ;
                         {ImplDefinition}
```

```
ImplDefinition        =  ImportDef                 // see A.2
                      |  FunctionDef               // see A.3
                      |  GraphDef                  // see A.3
                      |  MacroDef                  // see A.4
                      |  TypeDef                   // see A.5
                      |  ClassDef                  // see A.6
                      |  GenericsDef               // see A.7
                      |  ForeignExportDef          // see A.8
```

```
DefDefinition         =  ImportDef                 // see A.2
                      |  FunctionExportTypeDef     // see A.3
                      |  MacroDef                  // see A.4
                      |  TypeDef                   // see A.5
                      |  ClassExportDef            // see A.6
                      |  GenericExportDef          // see A.7
```

A.2 Import Definition

```
ImportDef             =  ImplicitImportDef
                      |  ExplicitImportDef
```

```
ImplicitImportDef     =  import [qualified] {ModuleName}-list ;
```

```
ExplicitImportDef     =  from ModuleName import [qualified] {Imports}-list ;
Imports               =  FunctionName
                      |  ::TypeName [ConstructorsOrFields]
                      |  class ClassName [Members]
                      |  instance ClassName {SimpleType}+
                      |  generic FunctionName
```

```
ConstructorsOrFields  =  (..)
                      |  ({ConstructorName}-list)
                      |  {..}
                      |  {{FieldName}-list}
Members               =  (..)
                      |  ({MemberName}-list)
```

A.3 Function Definition

```
FunctionDef           =  [FunctionTypeDef]
                         DefOfFunction

DefOfFunction         =  {FunctionAltDef ;}+
                      |  ABCCodeFunctionDef
FunctionAltDef        =  Function {Pattern}
                         {GuardAlt} {LetBeforeExpression} FunctionResult
                         [LocalFunctionAltDefs]
FunctionResult        =  =[>] FunctionBody
                      |  | Guard GuardRhs
GuardAlt              =  {LetBeforeExpression} | BooleanExpr GuardRhs
GuardRhs              =  {GuardAlt} {LetBeforeExpression} = [>] FunctionBody
                      |  {GuardAlt} {LetBeforeExpression} | otherwise GuardRhs

Function              =  FunctionName
                      |  (FunctionName)

LetBeforeExpression   =  #  {GraphDefOrUpdate}+
                      |  #!{GraphDefOrUpdate}+
GraphDefOrUpdate      =  GraphDef
                      |  Variable & {FieldName {Selection} = GraphExpr}-list ;
                      |  Variable & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list] ;

GraphDef              =  Selector =[:] GraphExpr ;
Selector              =  BrackPattern

Guard                 =  BooleanExpr
                      |  otherwise
BooleanExpr           =  GraphExpr

FunctionBody          =  RootExpression ;
                         [LocalFunctionDefs]

RootExpression        =  GraphExpr

LocalFunctionAltDefs  =  [where] { {LocalDef}+ }
LocalDef              =  GraphDef
                      |  FunctionDef
LocalFunctionDefs     =  [with] { {LocalDef}+ }

ABCCodeFunctionDef    =  Function {Pattern} = code [inline] { ABCInstructions }
```

A.3.1 Types of Functions

```
FunctionTypeDef       =  FunctionName :: FunctionType ;
                      |  (FunctionName) [FixPrec] [:: FunctionType] ;
FunctionType          =  [{ArgType}+ ->] Type [ClassContext] [UnqTypeUnEqualities]
ClassContext          =  | ClassConstraints {& ClassConstraints}
ClassConstraints      =  ClassOrGenericName-list {SimpleType}+
UnqTypeUnEqualities   =  [ {{UniqueTypeVariable}+ <= UniqueTypeVariable}-list ]
ClassOrGenericName    =  QClassName
                      |  FunctionName { | TypeKind | }

FunctionExportTypeDef =  FunctionName :: FunctionType [Special] ;
                      |  (FunctionName) [FixPrec] :: FunctionType [Special] ;
```

A.3.2 Patterns

```
Pattern              =  [Variable =:] BrackPattern
BrackPattern         =  PatternVariable
                     |  QConstructor
                     |  (GraphPattern)
                     |  SpecialPattern
                     |  DynamicPattern

PatternVariable      =  Variable
                     |  _

QConstructor         =  QConstructorName
                     |  (QConstructorName)

GraphPattern         =  QConstructor {Pattern}
                     |  GraphPattern QConstructorName GraphPattern
                     |  Pattern

SpecialPattern       =  BasicValuePattern
                     |  ListPattern
                     |  TuplePattern
                     |  ArrayPattern
                     |  RecordPattern
                     |  UnitPattern

BasicValuePattern    =  BasicValue
BasicValue           =  IntDenotation                 // see B.4
                     |  RealDenotation                // see B.4
                     |  BoolDenotation                // see B.4
                     |  CharDenotation                // see B.4

ListPattern          =  [[ListKind][{LGraphPattern}-list [: GraphPattern]] [SpineStrictness]]
ListKind             =  !                             // head strict list
                     |  #                             // head strict, unboxed list
                     |  |                             // overloaded list
SpineStrictness      =  !                             // tail (spine) strict list
LGraphPattern        =  GraphPattern
                     |  CharsDenotation               // see B.4

TuplePattern         =  (GraphPattern, {GraphPattern}-list)

RecordPattern        =  {[QTypeName |] {FieldName [= GraphPattern]}-list}

ArrayPattern         =  {{GraphPattern}-list}
                     |  {{ArrayIndex = Variable}-list}
                     |  StringDenotation

UnitPattern          =  ()

DynamicPattern       =  (GraphPattern :: DynamicType)
DynamicType          =  [UnivQuantVariables] {DynPatternType}+ [ClassContext]
DynPatternType       =  Type
                     |  TypePatternVariable
                     |  OverloadedTypePatternVariable
TypePatternVariable  =  Variable
OverloadedTypeVariable =  Variable^
```

A.3.3 Graph Expressions

```
GraphExpr            =  Application

Application          =  {BrackGraph}+
                     |  GraphExpr Operator GraphExpr
                     |  GenericAppExpr
Operator             =  QFunctionName                 // see A.9
                     |  QConstructorName              // see A.9
```

```
BrackGraph              =  GraphVariable
                        |  QConstructor
                        |  QFunction
                        |  (GraphExpr)
                        |  LambdaAbstr
                        |  CaseExpr
                        |  LetExpression
                        |  SpecialExpression
                        |  DynamicExpression
                        |  MatchesPatternExpr

GraphVariable           =  Variable                    // see A.9
                        |  SelectorVariable            // see A.9

QFunction               =  QFunctionName               // see A.9
                        |  (QFunctionName)             // see A.9

LambdaAbstr             =  \ {Pattern}+ {LambdaGuardAlt} {LetBeforeExpression} LambdaResult
LambdaResult            =  =  GraphExpr
                        |  -> GraphExpr
                        |  | Guard LambdaGuardRhs
LambdaGuardAlt          =  {LetBeforeExpression} | BooleanExpr LambdaGuardRhs
LambdaGuardRhs          =  {LambdaGuardAlt} {LetBeforeExpression} LambdaGuardResult
LambdaGuardResult       =  =  GraphExpr
                        |  -> GraphExpr
                        |  | otherwise LambdaGuardRhs

CaseExpr                =  case GraphExpr of
                           { {CaseAltDef}+ }
                        |  if BrackGraph BrackGraph BrackGraph
CaseAltDef              =  {Pattern}
                           {CaseGuardAlt} {LetBeforeExpression} CaseResult
                           [LocalFunctionAltDefs]
CaseResult              =  = [>] FunctionBody
                        |  -> FunctionBody
                        |  | Guard CaseGuardRhs
CaseGuardAlt            =  {LetBeforeExpression} | BooleanExpr CaseGuardRhs
CaseGuardRhs            =  {CaseGuardAlt} {LetBeforeExpression} CaseGuardResult
CaseGuardResult         =  = [>] FunctionBody
                        |  -> FunctionBody
                        |  | otherwise CaseGuardRhs

LetExpression           =  let { {LocalDef}+ } in GraphExpr

SpecialExpression       =  BasicValue
                        |  List
                        |  Tuple
                        |  Array
                        |  ArraySelection
                        |  Record
                        |  RecordSelection
                        |  UnitConstructor

List                    =  ListDenotation
                        |  DotDotExpression
                        |  ZF-expression
ListDenotation          =  [[ListKind] [{LGraphExpr}-list [: GraphExpr]] [SpineStrictness] ]
LGraphExpr              =  GraphExpr
                        |  CharsDenotation             // see B.4
DotDotExpression        =  [[ListKind] GraphExpr [, GraphExpr] . .[GraphExpr] [SpineStrictness] ]
ZF-expression           =  [[ListKind] GraphExpr \\ {Qualifier}-list [SpineStrictness]]
```

```
Qualifier       =  Generators {, let { {LocalDef}+ } } {|Guard}
Generators      =  Generator {& Generator}
Generator       =  Selector <- ListExpr                 // select from a lazy list
                |  Selector <|- ListExpr                // select from an overloaded list
                |  Selector <-: ArrayExpr               // select from an array
Selector        =  BrackPattern                         // for brack patterns see 3.2
ListExpr        =  GraphExpr
ArrayExpr       =  GraphExpr

Tuple           =  (GraphExpr,{GraphExpr}-list)

Array           =  ArrayDenotation
                |  ArrayUpdate
                |  ArrayComprehension
                |  ArraySelection
ArrayDenotation =  {[ArrayKind] {GraphExpr}-list}
                |  StringDenotation                     // see B.4
ArrayUpdate     =  { ArrayExpr & {ArrayIndex {Selection} = GraphExpr}-list [\\ {Qualifier}-list]}
ArrayComprehension = {[ArrayKind] GraphExpr \\ {Qualifier}-list}
ArraySelection  =  ArrayExpr.ArrayIndex {Selection}
                |  ArrayExpr!ArrayIndex {Selection}
Selection       =  .FieldName
                |  .ArrayIndex
ArrayExpr       =  GraphExpr
ArrayIndex      =  [{IntegerExpr}-list]
IntegerExpr     =  GraphExpr

Record          =  RecordDenotation
                |  RecordUpdate
RecordDenotation =  {[QTypeName|] {FieldName = GraphExpr}-list]}
RecordUpdate    =  {[QTypeName|][RecordExpr &][{FieldName {Selection} = GraphExpr}-list]}
RecordExpr      =  GraphExpr
RecordSelection =  RecordExpr [.QTypeName].FieldName {Selection}
                |  RecordExpr [.QTypeName]!FieldName {Selection}

UnitConstructor =  ()

DynamicExpression =  dynamic GraphExpr [:: [UnivQuantVariables] Type [ClassContext]]

MatchesPatternExpr =  GraphExpr =: QConstructorName { _ }
                   |  GraphExpr =: BrackPattern
```

A.4 Macro Definition

```
MacroDef        =  [MacroFixityDef]
                   DefOfMacro
MacroFixityDef  =  (FunctionName) [FixPrec] ;
DefOfMacro      =  Function {Variable} :== FunctionBody ;
                   [LocalFunctionAltDefs]
```

A.5 Type Definition

```
TypeDef         =  AlgebraicTypeDef
                |  RecordTypeDef
                |  SynonymTypeDef
                |  AbstractTypeDef
                |  AbstractSynonymTypeDef
                |  ExtensibleAlgebraicTypeDef
                |  AlgebraicTypeDefExtension
                |  NewTypeDef

AlgebraicTypeDef =  ::TypeLhs  = ConstructorDef
                               {| ConstructorDef} ;
ConstructorDef  =  [ExistQuantVariables] ConstructorName {ArgType} {& ClassConstraints}
                |  [ExistQuantVariables] (ConstructorName) [FixPrec] {ArgType} {& ClassConstraints}

TypeLhs         =  [*] TypeConstructor {[*]TypeVariable}
TypeConstructor =  TypeName             // see A.9
```

```
ExistQuantVariables   =  E.{TypeVariable }+:

FixPrec               =  infixl [Prec]
                      |  infixr [Prec]
                      |  infix [Prec]
Prec                  =  Digit                          // see A.9

BrackType             =  [Strict] [UnqTypeAttrib] SimpleType

Strict                =  !

UnqTypeAttrib         =  *
                      |  UniqueTypeVariable:    // see A.9
                      |  .

Type                  =  {BrackType}+
ArgType               =  BrackType
                      |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type [ClassContext])

UnivQuantVariables    =  A.{TypeVariable }+:

RecordTypeDef         =  ::TypeLhs = [ExistQuantVariables] [Strict] {{FieldName :: FieldType}-list} ;
FieldType             =  [Strict] Type
                      |  UnivQuantVariables [Strict] Type
                      |  [Strict] [UnqTypeAttrib] (UnivQuantVariables Type)

SynonymTypeDef        =  ::TypeLhs :== Type ;

AbstractTypeDef       =  :: [!][UnqOrCoercibleTypeAttrib] TypeConstructor {[*]TypeVariable};
UnqOrCoercibleTypeAttrib =  *
                      |  .

AbstractSynonymTypeDef =  AbstractTypeDef (:== Type ) ;

ExtensibleAlgebraicTypeDef =  ::TypeLhs = {ConstructorDef |} ..;

AlgebraicTypeDefExtension =  ::TypeLhs | ConstructorDef {| ConstructorDef} ;

NewTypeDef            =  ::TypeLhs =: ConstructorName SimpleType ;
```

A.5.1 Types Expression

```
SimpleType            =  TypeVariable                   // see A.9
                      |  QTypeName
                      |  (Type)
                      |  PredefinedType
                      |  PredefinedTypeConstructor

PredefinedType        =  BasicType
                      |  ListType
                      |  TupleType
                      |  ArrayType
                      |  ArrowType
                      |  PredefType

BasicType             =  Int
                      |  Real
                      |  Char
                      |  Bool

ListType              =  [[ListTypeKind] Type [SpineStrictness]]
ListTypeKind          =  !                              // head strict list
                      |  #                              // head strict, unboxed list

TupleType             =  ([Strict] Type,{[Strict] Type}-list)
```

```
ArrayType       =  {[ArrayKind] Type}
ArrayKind       =  !                    // strict array
                |  #                    // unboxed array

PredefType      =  World                // see StdWorld.dcl
                |  File                 // see StdFileIO.dcl
                |  String               // synonym for {#Char}
                |  ()                   // unit type constructor

PredefinedTypeConstructor
                =  []                   // list type constructor
                |  [! ]                 // head strict list type constructor
                |  [ !]                 // tail strict list type constructor
                |  [!!]                 // strict list type constructor
                |  [#]                  // unboxed head strict list type
                |  [#!]                 // unboxed strict list type
                |  ({,}+)               // tuple type constructor (arity >= 2)
                |  {}                   // lazy array type constructor
                |  {!}                  // strict array type constructor
                |  {#}                  // unboxed array type constructor
                |  (->)                 // arrow type constructor
```

A.6 Class and Instance Definitions

```
ClassDef        =  TypeClassDef
                |  TypeClassInstanceDef

TypeClassDef    =  class ClassName {[.]TypeVariable}+ [ClassContext]
                   [[where] { {ClassMemberDef}+ }] ;
                |  class FunctionName {[.]TypeVariable}+ :: FunctionType;
                |  class (FunctionName) [FixPrec] {[.]TypeVariable}+ :: FunctionType;

ClassMemberDef  =  FunctionTypeDef
                   [MacroDef]

TypeClassInstanceDef
                =  instance QClassName Type+ [ClassContext]
                   [where] { {FunctionDef}+ } ;

ClassExportDef  =  TypeClassDef
                |  TypeClassInstanceExportDef

TypeClassInstanceExportDef
                =  instance ClassName InstanceExportTypes ;
InstanceExportTypes
                =  {Type+ [ClassContext]}-list
                |  Type+ [ClassContext] [where] {{FunctionTypeDef}+ }
                |  Type+ [ClassContext] [Special]
Special         =  special {{TypeVariable = Type}-list { ; {TypeVariable = Type}-list }}
```

A.7 Generic Definitions

```
GenericsDef     =  GenericDef ;
                |  GenericCase;
                |  DeriveDef ;

GenericDef      =  generic FunctionName TypeVariable+ [GenericDependencies] :: FunctionType
GenericDependencies
                =  | {FunctionName TypeVariable+ }-list

GenericCase     =  FunctionName { | GenericTypeArg | } {Pattern}+ = FunctionBody
GenericTypeArg  =  GenericMarkerType [of Pattern]
                |  TypeName
                |  TypeVariable
GenericMarkerType
                =  CONS
                |  OBJECT
                |  RECORD
                |  FIELD

DeriveDef       =  derive FunctionName {DerivableType}-list
                |  derive class ClassName {DerivableType}-list
DerivableType   =  TypeName
                |  PredefinedTypeConstructor
```

```
GenericAppExpression  =  FunctionName { |TypeKind| } GraphExpr
TypeKind              =  *
                      |  TypeKind -> TypeKind
                      |  IntDenotation
                      |  (TypeKind)
                      |  { |TypeKind| }
```

```
GenericExportDef      =  GenericDef ;
                      |  derive FunctionName {DeriveExportType [UsedGenericDependencies]}-list ;
                      |  derive class ClassName {DerivableType}-list ;
                      |  FunctionName { |GenericExportTypeArg| } {Pattern}+ = FunctionBody
```

```
DeriveExportType      =  TypeName
                      |  GenericMarkerType [of UsedGenericInfoFields]
                      |  PredefinedTypeConstructor
                      |  TypeVariable
UsedGenericInfoFields =  {[{FieldName}-list]}
                      |  Variable
```

```
UsedGenericDependencies  =  with {UsedGenericDependency}
UsedGenericDependency    =  Variable
                         |  _
```

```
GenericExportTypeArg  =  GenericMarkerType [of Pattern]
                      |  UNIT | PAIR | EITHER
```

A.8 Foreign Export Definition

```
ForeignExportDef  =  foreign export [ ccall | stdcall ] FunctionName ;
```

A.9 Names

```
ModuleName           =  LowerCaseId  |  UpperCaseId  |  ModuleDirectoryName . ModuleName
ModuleDirectoryName  =  LowerCaseId  |  UpperCaseId
FunctionName         =  LowerCaseId  |  UpperCaseId  |  SymbolId
ConstructorName      =                 UpperCaseId  |  SymbolId
SelectorVariable     =  LowerCaseId
Variable             =  LowerCaseId
MacroName            =  LowerCaseId  |  UpperCaseId  |  SymbolId
FieldName            =  LowerCaseId
TypeName             =                 UpperCaseId  |  SymbolId
TypeVariable         =  LowerCaseId
UniqueTypeVariable   =  LowerCaseId
ClassName            =  LowerCaseId  |  UpperCaseId  |  SymbolId
MemberName           =  LowerCaseId  |  UpperCaseId  |  SymbolId
```

```
QFunctionName        =  QLowerCaseId  |  QUpperCaseId  |  QSymbolId
QConstructorName     =                  QUpperCaseId  |  QSymbolId
QTypeName            =                  QUpperCaseId  |  QSymbolId
QClassName           =  QLowerCaseId  |  QUpperCaseId  |  QSymbolId
```

Appendix B

Lexical Structure

In this appendix the lexical structure of CLEAN is given. It describes the kind of tokens recognised by the scanner/parser. In particular it summarizes the keywords, symbols and characters which have a special meaning in the language.

B.1 Lexical Program Structure

In this Section the lexical structure of CLEAN is given. It describes the kind of tokens recognised by the scanner/parser. In particular it summarizes the keywords, symbols and characters which have a special meaning in the language.

LexProgram	=	{ Lexeme | {Whitespace}+ }	
Lexeme	=	ReservedKeywordOrSymbol	// see Section B.5
	|	ReservedChar	// see Section B.4
	|	Literal	
	|	Identifier	
Identifier	=	LowerCaseId	// see A.9
	|	UpperCaseId	// see A.9
	|	SymbolId	// see A.9
Literal	=	IntDenotation	// see B.4
	|	RealDenotation	// see B.4
	|	BoolDenotation	// see B.4
	|	CharDenotation	// see B.4
	|	CharsDenotation	// see B.4
	|	StringDenotation	// see B.4

Whitespace	=	space	// a space character
	|	tab	// a horizontal tab
	|	newline	// a newline char
	|	formfeed	// a formfeed
	|	verttab	// a vertical tab
	|	Comment	// see Section B.2

B.2 Comments

Comment	=	// AnythingTillNL newline	
	|	/* AnythingTill/* Comment AnythingTill*/ */	// comments may be nested
	|	/* AnythingTill*/ */	
AnythingTillNL	=	{AnyChar/ newline}	// no newline
AnythingTill/*	=	{AnyChar/ /*}	// no "/*"
AnythingTill*/	=	{AnyChar/ */}	// no "*/"
AnyChar	=	IdChar	// see A.9
	|	ReservedChar	
	|	SpecialChar	

B.3 Identifiers

LowerCaseId	=	LowerCaseChar{IdChar}
UpperCaseId	=	UpperCaseChar{IdChar}
SymbolId	=	{SymbolChar}+

QLowerCaseId	=	[ModuleQualifier]LowerCaseId
QUpperCaseId	=	[ModuleQualifier]UpperCaseId
QSymbolId	=	[ModuleQualifier space]SymbolId
ModuleQualifier	=	`'ModuleName'.`

```
LowerCaseChar   = a | b | c | d | e | f | g | h | i | j | k | l | m
                | n | o | p | q | r | s | t | u | v | w | x | y | z
UpperCaseChar   = A | B | C | D | E | F | G | H | I | J | K | L | M
                | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
SymbolChar      = ~ | @ | # | $ | % | ^ | ? | ! | :
                | + | - | * | < | > | \ | / | | | & | =
IdChar          = LowerCaseChar
                | UpperCaseChar
                | Digit                                  // see B.4
                | _ | `
```

B.4 Denotations

```
IntDenotation    = [Sign]{Digit}+                        // decimal number
                 | [Sign]0{OctDigit}+                    // octal number
                 | [Sign]0x{HexDigit}+                   // hexadecimal number
Sign             = + | -
RealDenotation   = [Sign]{Digit}+.{Digit}+[E[Sign]{Digit}+]
BoolDenotation   = True | False
CharDenotation   = CharDel AnyChar/CharDel CharDel
StringDenotation = StringDel{AnyChar/StringDel}StringDel
CharsDenotation  = CharDel {AnyChar/CharDel}+ CharDel
```

```
AnyChar          = IdChar | ReservedChar | SpecialChar
ReservedChar     = (  | )  | {  | }  | [  | ]  | ;  | ,  | .
SpecialChar      = \n  | \r  | \f  | \b          // newline,return,formf,backspace
                 | \t  | \\  | \CharDel         // tab,backslash,character delimiter
                 | \StringDel                    // string delimiter
                 | \{OctDigit}+                  // octal number
                 | \x{HexDigit}+                 // hexadecimal number
                 | \IdChar                       // escape any other character
```

```
Digit            = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
OctDigit         = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
HexDigit         = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                 | A | B | C | D | E | F
                 | a | b | c | d | e | f
```

```
CharDel          = '
StringDel        = "
```

B.5 Reserved Keywords and Symbols

Below the keywords and symbols are listed which have a special meaning in the language. Some symbols only have a special meaning in a certain context. Outside this context they can be freely used if they are not a reserved character (see B.4). In the comment it is indicated for which context (name space) the symbol is predefined.

```
ReservedKeywordOrSymbol =
// in all contexts:
                   /*          // begin of comment block
                 | */          // end of comment block
                 | //          // rest of line is comment
                 | ::          // begin of a type definition
                 | :==         // in a type synonym or macro definition
                 | =           // in a function, graph, alg type, rec field, case, lambda
                 | =:          // labeling a graph definition
                 | =>          // in a function definition
                 | ;           // end of a definition (if no layout rule)
                 | foreign     // begin of foreign export
```

```
// in global definitions:
                 | from        // begin of symbol list for imports
                 | definition  // begin of definition module,
                 | implementation // begin of implementation module
                 | import      // begin of import list
                 | module      // in module header
                 | system      // begin of system module
```

// in function definitions:	->	// in a case expression, lambda abstraction
	[	// begin of a list
	:	// cons node
	]	// end of a list
	\\	// begin of list or array comprehension
	<-	// list gen. in list or array comprehension
	<-:	// array gen. in list or array comprehension
	{	// begin of a record or array, begin of a scope
	}	// end of a record or array, end of a scope
	.	// a record or array selector
	!	// a record or array selector (for unique objects)
	&	// an update of a record or array, zipping gener.
	case	// begin of case expression
	code	// begin code block in a syst impl. module
	if	// begin of a conditional expression
	in	// end of (strict) let expression
	let	// begin of let expression
	#	// begin of let expression (for a guard)
	#!	// begin of strict let expression (for a guard)
	of	// in case expression
	where	// begin of local def of a function alternative
	with	// begin of local def in a rule alternative

// in type specifications:	!	// strict type
	.	// uniqueness type variable
	#	// unboxed type
	*	// unique type
	:	// in a uniqueness type variable definition
	->	// function type constructor
	[],[!],[!!],[#],[#!]	// lazy list, head strict, strict, unboxed, unboxed strict
	[|]	// overloaded list
	(,),(,,),(,,,),...	// tuple type constructors
	{},{!},{#}	// lazy, strict, unboxed array type constr.
	infix	// infix indication in operator definition
	infixl	// infix left indication in operator definition
	infixr	// infix right indication in operator definition
	special	// to create a specialized instance
	Bool	// type Boolean
	Char	// type character
	File	// type file
	Int	// type integer
	Real	// type real
	World	// type world

// in class definitions:	class	// begin of type class definition
	instance	// def of instance of a type class
	derive	// derive instance of generic function

Appendix C

Bibliography

You can find all our papers on our site: http://wiki.clean.cs.ru.nl/Publications

Peter Achten, John van Groningen and Rinus Plasmeijer (1992). "High-level specification of I/O in functional languages". *In: Proc. of the Glasgow workshop on Functional programming*, ed. J. Launchbury and P. Sansom, Ayr, Scotland, Springer-Verlag, Workshops in Computing, pp. 1-17.

Peter Achten and Rinus Plasmeijer (1995). "The Ins and Outs of CONCURRENT CLEAN I/O". *Journal of Functional Programming,* 5, 1, pp. 81-110.

Peter Achten and Rinus Plasmeijer (1997). "Interactive Functional Objects in CLEAN". In: *Proc. of the 1997 Workshop on the Implementation of Functional Languages (IFL'97)*, ed. K. Hammond Davie, T., and Clack, C., St.Andrews, Scotland, pp. 387-406.

Artem Alimarine and Rinus Plasmeijer. A Generic Programming Extension for Clean. In: Arts, Th., Mohnen, M. eds. Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01, Älvsjö, Sweden, September 24-26, 2001, Ericsson Computer Science Laboratory, pp.257-278.

Tom Brus, Marko van Eekelen, Maarten van Leer, Rinus Plasmeijer (1987). 'CLEAN - A Language for Functional Graph Rewriting'. *Proc. of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, Portland, Oregon, USA, LNCS 274, Springer Verlag, 364-384.

Barendregt, H.P. (1984). The Lambda-Calculus, its Syntax and Semantics. North-Holland.

Henk Barendregt, Marko van Eekelen, John Glauert, Richard Kennaway, Rinus Plasmeijer, Ronan Sleep (1987). 'Term Graph Rewriting'. *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, part II, Eindhoven, The Netherlands. LNCS 259, Springer Verlag, 141-158.

Erik Barendsen and Sjaak Smetsers (1993a). 'Extending Graph Rewriting with Copying'. In: *Proc. of the Seminar on Graph Transformations in Computer Science*, ed. B. Courcelle, H. Ehrig, G. Rozenberg and H.J. Schneider, Dagstuhl, Wadern, Springer-Verlag, Berlin, LNCS 776, Springer Verlag, pp 51-70.

Erik Barendsen and Sjaak Smetsers (1993b). 'Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract)'. In: *Proc. of the 13th Conference on the Foundations of Software Technology & Theoretical Computer Science*, ed. R.K. Shyamasundar, Bombay, India, LNCS 761, Springer Verlag, pp. 41-51.

Bird, R.S. and P. Wadler (1988). *Introduction to Functional Programming*. Prentice Hall.

Marko van Eekelen, Rinus Plasmeijer, Sjaak Smetsers (1991). 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures'. In Kaplan, S. and M. Okada (Eds.) *Proc. of the 2nd Int. Worksh. on Conditional and Typed Rewriting Systems (CTRS'90)*, 1990. Montreal, Canada, LNCS 516, Springer Verlag, 354-370.

Eekelen, M.C.J.D. van, J.W.M. Smetsers, M.J. Plasmeijer (1997). "Graph Rewriting Semantics for Functional Programming Languages". In: *Proc. of the CSL '96, Fifth Annual conference of the European Association for Computer Science Logic (EACSL)*, ed. Marc Bezem Dirk van Dalen, Utrecht, Springer-Verlag, LNCS, 1258, pp. 106-128.

Harper, R., D. MacQueen and R. Milner (1986). 'Standard ML'. Edinburgh University, Internal report ECS-LFCS-86-2.

Hindley R. (1969). The principle type scheme of an object in combinatory logic. *Trans. of the American Math. Soc.*, **146**, 29-60.

Hudak, P. , S. Peyton Jones, Ph. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, Th. Johnsson, D. Kieburtz, R. Nikhil, W. Partain and J. Peterson (1992). 'Report on the programming language Haskell'. *ACM SigPlan notices,* 27, 5, pp. 1-164.

John van Groningen and Rinus Plasmeijer. Strict and unboxed lists using type constructor classes in a lazy functional language. Presented at the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Älvsjö, Sweden, September 24-26, 2001, Ericsson Computer Science Laboratory.

Jones, M.P. (1993). *Gofer - Gofer 2.21 release notes*. Yale University.

Marko Kesseler (1991). 'Implementing the ABC machine on transputers'. In: *Proc. of the 3rd International Workshop on Implementation of Functional Languages on Parallel Architectures*, ed. H. Glaser and P. Hartel, Southampton, University of Southampton, Technical Report 91-07, pp. 147-192.

Kesseler, M.H.G. (1996). The Implementation of Functional Languages on Parallel Machines with Distributed Memory. Ph.D., University of Nijmegen.

Milner, R.A. (1978). 'Theory of type polymorphism in programming'. *Journal of Computer and System Sciences,* 17, 3, 348-375.

Mycroft A. (1984). Polymorphic type schemes and recursive definitions. In *Proc. International Conference on Programming,* Toulouse (Paul M. and Robinet B., eds.), LNCS 167, Springer Verlag, 217 239.

Eric Nöcker, Sjaak Smetsers, Marko van Eekelen, Rinus Plasmeijer (1991). 'CONCURRENT CLEAN'. In Aarts, E.H.L., J. van Leeuwen, M. Rem (Eds.), *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'91)* , Vol II, Eindhoven, The Netherlands, LNCS 505, Springer Verlag, June 1991, 202-219.

Eric Nöcker (1993). 'Strictness analysis using abstract reduction'. In: *Proc. of the 6th Conference on Functional Programming Languages and Computer Architectures*, ed. Arvind, Copenhagen, ACM Press, pp. 255-265.

Eric Nöcker and Sjaak Smetsers (1993). 'Partially strict non-recursive data types'. *Journal of Functional Programming,* **3**, 2, pp. 191-215.

Pil, M.R.C. (1999), Dynamic types and type dependent functions , In Proc. of Implementation of Functional Languages (IFL '98), London, U.K., Hammond, Davie and Clack Eds., Springer-Verlag, Berlin, Lecture Notes in Computer Science 1595, pp 169-185.

Rinus Plasmeijer and Marko van Eekelen (1993). *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, ISBN 0-201-41663-8.

Sjaak Smetsers, Eric Nöcker, John van Groningen, Rinus Plasmeijer (1991). 'Generating Efficient Code for Lazy Functional Languages'. In Hughes, J. (Ed.), *Proc. of the Fifth International Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, USA, LNCS 523, Springer Verlag, 592-618.

Ronan Sleep, Rinus Plasmeijer and Marko van Eekelen (1993). *Term Graph Rewriting - Theory and Practice*. John Wiley & Sons.

Yoshihito Toyama, Sjaak Smetsers, Marko van Eekelen and Rinus Plasmeijer (1993). 'The functional strategy and transitive term rewriting systems'. In: *Term Graph Rewriting*, ed. Sleep, Plasmeijer and van Eekelen, John Wiley.

Turner, D.A. (1985). 'Miranda: a non-strict functional language with polymorphic types'. In: *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, ed. J.P. Jouannaud, Nancy, France. LNCS 201, Springer Verlag, 1-16.

Martijn Vervoort and Rinus Plasmeijer (2002). Lazy Dynamic Input/Output in the lazy functional language Clean - early draft -. In: Peña, R. ed. Proceedings of the 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Madrid, Spain, September 16-18, 2002, Technical Report 127-02, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, pages 404-408.

Arjen van Weelden and Rinus Plasmeijer (2002). Towards a Strongly Typed Functional Operating System. In: Peña, R. ed. Proceedings of the 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Madrid, Spain, September 16-18, 2002, Technical Report 127-02, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, pages 301-319.

Appendix D

Compiler Extensions

This appendix lists extensions and modifications to Clean that are supported by a compiler. Unfortunately they are not yet described in the previous chapters and appendices of this manual.

D.1 Clean 3.0 Compiler Extensions

D.1.1 New imports

- Identifiers can be imported qualified by adding `qualified` after `import` in the import statement. For example:

```
import qualified StdList;

from StdList import qualified drop,++;
```

Identifiers imported in this way can be used by prefixing the identifier with the module name between single quotes and a dot. If an identifier consists of special characters (for example ++) an additional single space is required between the dot and the identifier.

For example:

```
f l = 'StdList'.drop 1 (l 'StdList'. ++ [2]);
```

Currently field names of records are not imported by an implicit qualified import, but can be imported with an explicit qualified import.

Qualified names may only be used if a qualified import is used, not if the identifier is only imported by a normal (unqualified) import. An identifier may be imported both unqualified and qualified.

Qualified imports may be used in definition modules, but qualified identifiers cannot be imported from a (definition) module.

D.1.2 Uniqueness typing additions

- Updates of unique array elements:

A unique array element of a (unique) array of unique elements can be selected and updated, if the selection (using `![ ]`) and update (with the same index) occur in the same function and the array is not used in between (only the selected element is used).

For example, below a unique row is selected, updated by `inc_a` and finally the row of the array is updated.

```
inc_row :: !*{#*{#Int}} !Int -> *{#*{#Int}};
inc_row a row_i
    # (row,a) = a![row_i];
      row = inc_a 0 row;
    = {a & [row_i]=row};

inc_a :: !Int !*{#Int} -> *{#Int};
inc_a i a
    | i<size a
        # (n,a) = a![i];
          a & [i]=n+1;
        = inc_a (i+1) a;
        = a;
```

D.1.3 Hierarchical modules

The module name can be used to specify the directory containing the module. In that case the module name is the list of folder names of the directory, separated by `.`'s, followed by a `.` and the file name. For example the implementation module `X.Y.Z` is stored in file `X/Y/Z.icl` (file `Z.icl` in subfolder `Y` of folder `Z`). The path containing the first folder (`X` in this case) should be a module search path for the compiler.

D.2 Clean Development Compiler Extensions

- type GenericInfo (in module StdGeneric) changed.
- generic instances of generic representation types (e.g. CONS) may occur in definition modules.
- builtin Maybe types added (?, ?^, ?# with constructors ?Just, ?None, ?^Just, ?^None, ?#Just, ?#None).

D.3 Clean ITask Compiler Extensions

- Function arguments may have contexts with universally quantified type variables.
- Constructors may have contexts (for normal and universally quantified type variables).
- dynamic types may have contexts.