Part I
Chapter 4
The Power of Types

Clean is a strongly typed language. This means that every expression in the language has a type and that type correctness can be verified before the program is executed. Types are deduced and checked by the compiler. Ill-typed programs are not accepted. Many errors can be found and reported at compile time thanks to the checks performed type system.

Type systems can also be used to increase the expressive power of a language. In this chapter a number of language features which are related to the type system are explained. First we will explain the overloading mechanism of Clean which makes it possible to use the same function name for different functions performing similar kind of actions. It can be used to write (parts of) programs in such a way that the actual data structures being used can be chosen in a later state of the design (section 4.1). Then we explain how one can store objects of different types into a recursive data structure like a list using existentially quantified data types. In this way an object oriented style of programming can be achieved (section 4.2). Finally we treat an important feature of Clean: the uniqueness type system (section 4.3). It makes it possible to destructively update data structures like arrays and files without violating the pure functional semantics of the language.

4.1 Type Classes

In the previous chapters we have seen the type rules for functions and applied these rules in the functions shown. A summary of the type rules is:

- each function argument should be used with the same type at any occurrence in the function body;
- the type of the function should be an instance of the type of each function alternative;
- the type of all function bodies within an alternative should be an instance of the type of the function alternative, and hence an instance of the type of the function;
- the number of arguments op the function should be equal to the number of arguments of the type.

When the programmer specifies a type for a function it should be consistent with these rules. An *instance* of a type `t` is obtained by replacing variables in the type `t` by other types. A function is called *polymorph* when there occur type variables in its type. This indicates that the function can be applied to arguments of many different types. When there are no restrictions on the type variable the function can even be applied to arguments of any type. Some well-known examples of polymorphic typed functions are:

```
id :: t -> t
id x = x
```

```
hd :: [t] -> t
hd [x:_] = x

if :: Bool t t -> t
if c t e | c = t
             = e
```

In the functions `id`, `hd` and `if` any type can be used for `t`.

4.1.1 Overloading

A polymorphic function is defined over a range of types, acting in the same way for each concrete type. Another mechanism which allows functions to be applied with different concrete types is called *overloading* or *ad-hoc polymorphism*. Overloading occurs when a set of functions is defined. Each of these functions has a different type, but all function in this class have the same name. So, one (overloaded) function name (e.g. `+`) is associated with different operations (`Int` addition, `Real` addition, *etcetera*).

Usually it is considered an error when different functions have the same name. However, when a number of functions perform similar actions to different data types, it can be convenient to given them all the same name. The definition of an overloaded function consists of two parts:

· the *signature* of the overloaded function, i.e. a name and type specification;

· a collection of (type dependent) concrete realizations; the so called *instances*.

For reasons of flexibility, most programming languages allow those parts to be specified separately. In Clean, a signature is introduced by a *class declaration*. This class declaration tells the Clean system that it is possible that there occur a number of functions with the given name. In order to guarantee that these functions are sufficient similar, the type of these functions should be an instance of the type given in the signature. Examples of such signatures are the following (pre-defined) class declarations introducing some common overloaded operators.

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

class (*) infixl 7  a :: !a !a  -> a
class (/) infix 7   a :: !a !a  -> a
class one           a ::  a

class (==) infix 2  a :: !a !a  -> Bool
class (<) infix 2   a :: !a !a  -> Bool
```

In each class declaration, one of the type variables appearing in the signature is denoted explicitly. This, so called *class variable* is used to relate the type of an overloaded operator to all the types of its instances. The latter are introduced by *instance declarations*. An instance declaration associates a function body with a concrete instance type. The type of this function is determined by substituting the instance type for the class variable in the corresponding signature. For example, we can define an instance of the overloaded operator `+` for strings, as follows.

```
instance + String
where   (+) s1 s2 = s1 +++ s2
```

By substituting `String` for `a` in the signature of `+` one obtains the type for the newly defined operator, to wit `!String !String -> String`. In Clean it is permitted to specify the type of an instance explicitly, provided that this specified type is exactly the same as the type obtained via the above-mentioned substitution. Among other things, this means that the following instance declaration is valid.

```
instance + String
where   (+) :: !String !String -> String
        (+) s1 s2 = s1 +++ s2
```

It is also possible to define instances of functions for polymorphic data types. In chapter I.3.1 we have shown the definition of the operators == and < for lists. Some other examples are:

```
instance + (x,y) | + x & + y
where   (+) (a,b) (x,y) = (a+x,b+y)

instance == (x,y) | == x & == y
where   (==) (a,b) (x,y) = a == x && b == y
```

In fact, a large number of these operators and instances for the basic types and data types are defined in `StdEnv`. In order to limit the size of the standard library only those operations that are considered the most useful are defined. It might happen that you have to define some instances of standard functions and operators yourself.

Observe that, what we have called an overloaded function is not a real function in the usual sense: An overloaded function actually stands for a whole family of functions. If an overloaded function is applied in a certain context, the type system determines (if possible) which concrete instance has been used. For instance, if we define

```
increment n = n + 1
```

it is clear that the `Int` addition is meant leading to a substitution of this `Int` version for `+`. However, it is often impossible to derive the concrete version of an overloaded function from the context in which it is applied. Consider the following definition:

```
double n = n + n
```

Now, one cannot determine which instance of `+` is meant. In fact, the function `double` becomes overloaded itself, which is reflected in its type:

```
double :: a -> a | + a
```

The *type context* `+ a` indicates the restriction that double is defined only on those objects that can be handled by a `+`. In general, a type context of the form `C a`, restricts instantiation of `a` to types for which an instance declaration of `C` exists. If a type context for `a` contains several class applications, it assumed that `a` is chosen from the instances types all these classes have in common.

One can, of course, use a more specific type for the function `double`. E.g.

```
double :: Int -> Int
double n = n + n
```

Obviously, `double` is not overloaded anymore: due to the additional type information, the instance of `+` to be used can now be determined.

Type contexts can become quite complex if several different overloaded functions are used in a function body. Consider, for example, the function `determinant` for solving quadratic equations.

```
determinant a b c = b * b - (fromInt 4) * a * c
```

The type of `determinant` is

```
determinant :: a a a -> a | *, -, fromInt a
```

To enlarge readability, it is possible to associate a new (class) name with a set of existing overloaded functions. E.g.

```
class Determinant a | *, -, fromInt a
```

The class `Determinant` consists of the overloaded functions `*`, `-` and `fromInt`. Using the new class in the type of `determinant` leads to:

```
determinant :: a a a -> a | Determinant a.
```

Notice the difference between the function `determinant` and the class `Determinant`. The class `Determinant` is just a shorthand notation for a set of type restriction. The name of such a type class should start with an uppercase symbol. The function `determinant` is just a function using the class `Determinant` as a compact way to define some restrictions on its type. As far as the Clean system is concerned it is a matter of coincidence that you find these names so similar.

Suppose `C1` is a new class, containing the class `C2`. Then `C2` forms a so called *subclass* of `C1`. 'Being a subclass of' is a transitive relation on classes: if `C1` on its turn is a subclass of `C3` then also `C2` is also a subclass of `C3`.

A class definition can also contain new overloaded functions, the so-called *members* of the class. For example, the class `PlusMin` can be defined as follows (assuming that `+`, `-` and `zero` are defined elsewhere).

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

To instantiate `PlusMin` one has to specify an instance for each of its members. For example, an instance of `PlusMin` for `Char` might look as follows.

```
instance PlusMin Char
where   (+) x y = toChar ((toInt x) + (toInt y))
        (-) x y = toChar ((toInt x) - (toInt y))
        zero    = toChar 0
```

Some of the readers will have noticed that the definition of an overloaded function is essentially the same as the definition of a class consisting of a single member. Indeed, classes and overloaded operators are one and the same concept. Since operators are just functions with two arguments, you can use operators in type classes in the same way as ordinary functions.

A class defines in fact a family of functions with the same name. The difference between the functions is the type of the arguments. For an polymorph function one and the same definition is used for all argument types. For an overloaded function (a class member) a separate function is used for each type of instance defined. In order to guarantee that only a single instance is defined for each type, it is not allowed to define instances for type synonyms. The selection of the instance of the overloaded function to be applied is done by the Clean system based on type information. Whenever possible this selection is done at compile-time. In some (rare) circumstances it is not possible to do this selection at compile-time. In those circumstances the selection is done at run-time.

In Clean, the general form of a class definition is a combination of the variants discussed so far: A new class consists of a collection of existing classes extended with a set of new members. Besides that, such a class will appear in a type context of any function that uses one or more of its members, of which the actual instance could not be determined. For instance, if the `PlusMin` class is used (instead of the separate classes `+`, `-` and `zero`), the types of `double` and `determinant` can be specified as:

```
double :: a -> a | PlusMin a
determinant :: a a a -> a | *, PlusMin, fromInt a
```

The Clean system itself is able to derive this kind of types with class restrictions.

The class `PlusMin` is defined in the standard environment (`StdClass`) is slightly different from the definition shown in this section. The definition is the standard environment is:

```
class PlusMin a | + , - , zero a
```

When you use the class `PlusMin` there is no difference between both definitions. However, when you define a new instance of the class you hav to be aware of the actual definition of the class. When the class contains members, you create an instance of the class as shown here. For a class that is defined by a class context, as `PlusMin` from `StdClass`, you define an instance by defining instances for all classes listed in the context. In the next section we show an example of the definition of an instance of this class.

4.1.2 A class for Rational Numbers

In chapter I.3 we introduced a type `Q` for representing rational numbers. These numerals are records consisting of a numerator and a denominator field, both of type `Int`:

```
:: Q    =   {   num :: Int
            ,   den :: Int
            }
```

We define the usual arithmetical operations on `Q` as instances of the corresponding type classes. For example,

```
instance + Q
where   (+) :: !Q !Q -> Q
        (+) x y = mkQ (x.num * y.den + x.den * y.num) (x.den * y.den)

instance - Q
where   (-) x y = mkQ (x.num * y.den - x.den * y.num) (x.den * y.den)

instance fromInt Q
where   fromInt i = mkQ i 1

instance zero Q
where   zero = fromInt 0
```

Using:

```
mkQ :: x x -> Q | toInt x
mkQ n d = simplify {num = toInt n, den = toInt d}

simplify :: Q -> Q
simplify {num=n,den=d}
    | d == 0    = abort "denominator of Q is 0!"
    | d < 0     = {num = ~n / g, den = ~d / g}
    | otherwise = {num =  n / g, den =  d / g}
where   g = gcd n d
```

At first sight, it seems as if the definition of, for example, the instance for `+` is recursive, for, an application of `+` also appears in the body of this instance. However, from its context, it immediately follows that the actual operation that is meant is the `+` for values of type `Int`.

When a new data type is introduced, it is often convenient if a string representation of this data type is available. Amongst other things, this representation can be used for printing a concrete value of that type on the screen. For this purpose, we introduce the class `toString`:

```
class toString a :: !a -> String
```

The corresponding instance of `toString` for `Q` might look as follows.

```
instance toString Q
where toString q
        | sq.den==1 = toString sq.num
        | otherwise = toString sq.num +++ "/" +++ toString sq.den
      where
        sq = simplify q
```

The execution of the program

```
Start :: String
Start = toString sum
where   sum :: Q
        sum = zero + zero
```

results in the string `"0"`.

It seems as if it makes no difference if we would write

```
Start = toString (zero + zero)
```

However, in this situation it is not possible to determine the used instances of `zero`, `+` and `toString` uniquely, i.e. there are several concrete instances that can be applied. The problem is that the expression `toString (zero + zero)` is *internally* overloaded: the overloading is not reflected by its result type (which is simply `String`). Such an expression will cause the compiler to generate the error message:

```
Type error [...]: "zero" (internal) overloading is insolvable
```

When it is known which instance of, for example, `zero` should be used, one can deduce the concrete instances of `+` and `toString`. Internal overloading can always be solved by introducing auxiliary local definitions that are typed explicitly (like the `sum` function in the above example).

Another way to solve the ambiguity is to indicate one of the instancesof the class as the *default instance*. In all situations where the overloading cannot be resolved, this default instance will be used. For instance, we can define the instance of type `Q` the default for the class `zero` by writing:

```
instance zero Q default
where zero = mkQ 0 1
```

Now it is allowed to write

```
Start = toString (zero + zero)
```

The context still does not determine which `zero` is used here, but now the Clean system picks the default one: the `zero` of type `Q`.[1]

By defining an instance of class `Enum` for the type `Q` it is even possible to generate list of rational numbers using dotdot expressions. Apart form the functions `+`, `-`, `zero` and `one`, the class `Enum` contains the ordering operator `<`. A suited instance declaration of `<` for `Q` is

```
instance < Q
where   (<) x y = x.num * y.den < x.den * y.num
```

A program like

```
Start :: [String]
Start = [toString q \\ q <- [zero, mkQ 1 3 .. mkQ 3 2]]
```

is type correct. It's execution yields:

```
["0","1/3","2/3","1/1","4/3"]
```

4.1.3 Derived class members

Sometimes, a member of a class is not really a new function, but defined in terms of other members (either of the same class or of a subclass). The standard environment, for example, introduces the class `Eq` containing the comparison operators `==` (already defined as class in `StdOverloaded`) and `<>` in the following way.

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

The `<>` operator is an example of , what is called, a *derived class member*: a member of which the body is included in the class definition itself (Clean considers derived members as macros[2]). In contrast to other functional languages, like Haskell and Gofer, the instances of derived members are never specified in Clean; they are *inherited* from the classes corresponding to the used operators (`==` in the above example).

In the same style we can define a complete set of ordering operators based on the operator `<`.

```
class Ord a | < a
where   (>)  infix 2 :: !a !a -> Bool | Ord a
        (>)  x y :== y < x

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

[1]In Clean 1.2 it is required that all overloaded functions involved have the same default type specified.

[2]In Clean 1.1 derived members are macros. A drawback of using macros to define derived members is that they cannot be used as a curried function. This is fixed in Clean 1.2.

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

In fact, also the equality operator == could be defined as a derived member, e.g. by specifying

```
    class Eq a | < a
    where   (==) infix 2 :: !a !a -> Bool | Eq a
            (==) x y :== x <= y && x >= y

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

By this mechanism, one obtains all ordering operations for a certain type, solely by defining an instance of < for this type. For efficiency reasons this is not done in the standard environment of Clean. In order to enable all possible comparision for some type `T` you should define an instance of < and ==.

When defining instances of functions acting on polymorphic data structures, these instances are often overloaded themselves, as shown by the following example.

```
    instance < [a] | < a
    where   (<) :: [a] [a] -> Bool | < a
            (<) _      []     = False
            (<} []     _      = True
            (<) [a:as] [b:bs] = a < b || a == b && as < bs
```

The instance type `[a]` is supplied with a type context which reflects that, in the body of the instance, the < operator is applied to the list elements. Observe that the specified type is, as always, the same as the type obtained from the signature of < after substituting `[a] | < a` for the class variable.

This example clearly shows the expressive power of the type classes. Suppose an instance < for some type `T` is available. With one single instance definition it possible to compare objects of type `[T]`, of type `[[T]]` and so on.

4.1.4 Type constructor classes

Until now, we assumed that each type constructor has a fixed arity indicating the number a type arguments, an application of that type constructor is supposed to have. For example the list constructor `[]` has arity 1, the 3-tuple constructor `(,,)` has arity 3, *etcetera*. *Higher-order types* are obtained by allowing type constructor applications in which the actual number of type arguments is less than the arity of the used constructor. In Clean it is possible to define classes with class variables ranging over such higher-order types. This leads to so-called *type constructor classes*. Type constructor classes can be used to define collections of overloaded higher-order functions. To explain the idea, consider the `map` function, defined as usual.

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

Experienced programmers will recognize that similar functions are often used for a wide range of other, mostly polymorphic data structures. E.g.

```
    Tree a = Node a [Tree a]

    mapTree :: (a -> b) (Tree a) -> Tree b
    mapTree f (Node el ls) = Node (f el) (map (MapTree f) ls)

    MayBe a = Just a | Nothing

    MapMayBe :: (a -> b) (MayBe a) -> MayBe b
    MapMayBe f (Just a) = Just (f a)
    MapMayBe f Nothing  = Nothing
```

Since all of these variants for `map` have the same kind of behaviour, it seems to be attractive to define them as instances of a single overloaded `map` function. Unfortunately, the overloading mechanism presented so far is not powerful enough to handle this case. For, an adequate class definition should be able to deal with (at least) the following type specifications:

```
(a -> b) [a] -> [b]
(a -> b) (Tree a) -> Tree b
(a -> b) (MayBe a) -> MayBe b.
```

It is easy to see, that a type signature for `map` such that all these type specifications can be obtained via the substitution of a single class variable by appropriate instance types, is impossible. However, by allowing class variables to be instantiated with higher-order instead of first-order types, such a type signature can be found, as indicated by the following class definition.

```
class map t :: (a -> b) (t a) -> t b
```

Here, the ordinary type variables `a` and `b` range over first-order types, whereas the class variable `t` ranges over higher-order types. To be more specific, the concrete instance types that can be substituted for `t` are (higher-order) types with one argument too few. The instance declarations that correspond to the different versions of `map` can now be specified as follows.

```
instance map []
where   map f l = [f e \\ e <- l]

instance map Tree
where   map f (Node el ls) = Node (f el) (map (map f) ls)

instance map MayBe
where   map f (Just a) = Just (f a)
        map f Nothing  = Nothing
```

The following instance declaration for map is also valid.

```
instance map (,) a
where   map :: (a -> b) (c,a) -> (c,b)
        map f (a,b) = (a,f b)
```

Here `(,) a` denotes the 2-tuple type constructor applied to a type variable `a`. Observe that an instance for type `(,)` (i.e. the same type constructor, but now with no type arguments) is impossible.

4.2 Existential types[3]

Polymorphic algebraic data types offer a large flexibility when building new data structures. For instance, one can use one and the same type definition for lists of integers, for list of characters, or even for lists of lists of something. However, the types of the objects stored in such a data structures are fixed, e.g. a list cannot contain both integers and characters. Of course, one can solve this problem *ad hoc*, e.g. by introducing the following auxiliary type.

```
:: OneOf a b = A a | B b
```

Indeed, a list of type `[OneOf Int Char]` may contain integers as well as characters, but again the choice is limited. In fact, the amount of freedom is determined by the number of type variables appearing in the data type definition.

To enlarge applicability, Clean has been extended with the possibility to use so called *existentially quantified* type variables (or, for short, *existential* type variables) in data type definitions. In the following example, we illustrate the use of existential variables by defining a list data structure elements of different types can be stored.

```
:: List E.a = Cons a (List Void) | Nil
```

[3]The syntax of existential types has been slightly changed in Clean version 1.2. This section still uses Clean 1.1 syntax.

The `E` prefix of `a` indicates that `a` is an existential type variable. In contrast to ordinary polymorphic (or, sometimes, called *universally quantified*) type variables, an existential type variable can be instantiated with a concrete type only at the moment a data object of the type in question is created. Consider, for example, the function

```
newlist = Cons 1 Nil
```

Here, the variable `a` of the constructor `Cons` is instantiated with `Int`. This concrete type information, however, is *hidden* by typing the result with type `List Void`. This `Void` type can be used independently of the actual type substitution. In fact, `Void` is the only type that is permitted on such vacant type argument positions, and moreover it is not allowed to use `Void` anywhere else in a type specification. By hiding the actual element types, one is able to build more complex structures like `Cons 1 (Cons 'a' Nil)`.

Existential type variables are not allowed in type specifications of functions, so data constructors are the only symbols with type specifications in which these special type variables may appear. If such data symbols are used in a pattern of a function, the corresponding existential type variables are never instantiated during type derivation of that function. Therefore, the following function `Hd` which yields the head element of a list

```
Hd (Cons hd tl) = hd
```

is illegal, for, the existential variable associated with `hd` is instantiated with the result type of `Hd`. The instance restriction becomes obvious if one realizes that the concrete types of the list elements is lost, and yielding an object with an unknown type may result in a program that is not type safe anymore. Returning to the list example, accessing the tail of the above list, e.g. by defining

```
Tl (Cons hd tl) = tl
```

is harmless: one can not do anything with `Tl`'s result that might disturb type safety.

Creating objects by existential types

Clearly, a data structure with existential quantified parts is not very useful if there exist no way of accessing the stored objects. For this reason, one usually provides such a data structure with an *interface*: a collection of functions for changing and/or retrieving information of the hidden object. So, the general form of these data structures is

```
:: Object E.a  =   { state :: a
                   , method_1 :: ... a ... -> ...
                   , method_2 :: ... -> ...a...
                   ,...
                   }
```

Those who are familiar with object oriented programming will recognize the similarity between the concept of *object-oriented data abstraction* and existentially quantified data structures in Clean.

We will illustrate the use of existentially quantified data structures with the aid of an example in which 'drawable objects' are represented as follows

```
:: Drawable E.a =  { state :: a
                   , move  :: Point -> a -> a
                   , draw  :: a Picture -> Picture
                   }
```

A `Drawable` contains a `state` field (e.g. the representation of a point, a line, a circle, *etcetera*) and two functions `move` and `draw` for moving and displaying the object on the screen, respectively. We use a number of graphical data types that are defined in the stadard I/O library in the module `Picture`. Drawing pictures is explained in more detail in the chapters I.5 and II.4.

```
:: Point       :== (!Int, !Int);
:: Line        :== (!Point, !Point);
:: Rectangle   :== (!Point, !Point);       // bounding points
:: Oval        :== Rectangle;              // bounding rectangle
:: Curve       :== (!Oval, !Int, !Int);    // Oval with start and end angle
```

First, we define two auxiliary function for creating the basic objects `Line` and `Curve`. The corresponding drawing routines are taken from the standard I/O library `Picture`; moving is defined straightforwardly. We use `+` for tuples from as defined in 4.1.1.

```
MakeLine :: Line -> Drawable Void
MakeLine line
        =   { state = line
            , move  = \dist line -> line + (dist,dist)
            , draw  = DrawLine
            }

MakeCurve :: Curve -> Drawable Void
MakeCurve curve
        =   { state = curve
            , move  = \dist (rect,a1,a2) -> (rect + (dist,dist),a1,a2)
            , draw  = DrawCurve
            }
```

A `Rectangle` is defined as a compound structure consisting of 4 lines, whereas a `Wedge` consists of 2 lines and a curve.

```
MakeRectangle :: Rectangle -> Drawable Void
MakeRectangle ((x1,y1),(x2,y2))
        =   { state = [ MakeLine ((x1,y1),(x1,y2)), MakeLine ((x1,y2),(x2,y2))
                      , MakeLine ((x2,y2),(x2,y1)), MakeLine ((x2,y1),(x1,y1))
                      ]
            , draw  = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
            , move  = \d -> map (MoveDrawable d)
            }

MakeWedge:: Curve -> Drawable Void
MakeWedge curve=:((begp,endp), a1, a2)
        =   { state = [ MakeLine (mid,mid+ epc1), MakeLine (mid,mid+ epc2)
                      , MakeCurve curve
                      ]
            , draw  = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
            , move  = \d -> map (MoveDrawable d)
            }
where   mid = (begp+endp)/(2,2)
        (epc1,epc2) = EndPointsOfCurve curve
```

Using a suitable implementation of `EndPointsOfCurve` and:

```
MoveDrawable :: Point (Drawable Void) -> Drawable Void
MoveDrawable p d=:{state,move} = {d & state = move p state}
```

Observe that moving and drawing of both compound objects is done in the same way. Moreover, due to the fact that (possibly different) `Drawables` can be stored in one list (for, the state of such objects is hidden) one can use standard list manipulating functions, such as `map` and `foldl` to perform these operations. Of course, the `Drawable` type is much too simple for being really useful: other functions have to added, e.g. predicates for testing whether a given point lies on the border, in the interior of a drawable or outside of it. Such extensions might be far from trivial, but nevertheless, the elegance of this method based on existentially quantified data structures is maintained. A full fledged type `Drawable` is developed in chapter II.4.

Using an ordinary polymorphic record for `Drawable`, instead of the existential type, is possible for the functions `MakeLine` and `makeCurve`. Their types become `Drawable Line` and `Drawable Curve` respectively. So, using a polymorphic record it is impossible to store these objects in one list as is done in the function `MakeWedge`.

Without existential types drawable objects can be modelled by an algebraic data type.

```
:: AlgDrawable
    = Line  Line
    | Curve Curve
    | Rect  [Line]
    | Wedge [AlgDrawable]
```

The manipulation of drawable objects is done by separately defined functions. These functions use detailed knowledge of the exact construction of the various alternatives of the algebraic data type `AlgDrawable`. Fortunately, the compiler generates a warning (`Function may fail`) when we accidentally omit one of the Constructors of `AlgDrawable`.

```
move :: Point AlgDrawable -> AlgDrawable
move p object
 = case object of
    Line  line         -> Line  (line + (p,p))
    Curve (rect,a1,a2) -> Curve (rect + (p,p),a1,a2)
    Rect  lines        -> Rect  [line + (p,p) \\ line <- lines]
    Wedge parts        -> Wedge (map (move p) parts)

draw :: AlgDrawable -> (Picture -> Picture)
draw object
 = case object of
    Line  line  -> DrawLine line
    Curve curve -> DrawCurve curve
    Rect  lines -> seq (map DrawLine lines)
    Wedge parts -> seq (map draw parts)
```

Although this is a way to handle objects that come in several different sorts, it has some drawbacks. The first disadvantage is that the properties and manipulation of drawable objects is distributed over a number of functions. For complicated types that are handled by many functions it becomes problematic to gather all information of that object. A second disadvantage of using an algebraic data type is that it becomes difficult to maintain the code. When an additional object like `Oval` is introduced, it is difficult to be sure that all necessary manipulation functions are updated to handle ovals.

When we change our mind and want to store a rectangle by its bounding points this is a very local change in the object-oriented approach. Only the function `MakeRectangle` needs to be changed:

```
MakeNewRectangle :: Rectangle -> Drawable Void
MakeNewRectangle rect
        =   { state = rect
            , move  = \p (p1,p2) -> (p1+p,p2+p)
            , draw  = DrawRectangle
            }
```

When we use the algebraic data type `AlgDrawable` to represent drawable objects and we want to implement the equivalent change, we have to change the data type and the corresponding manipulation functions.

```
:: AlgDrawable  = Line  Line
                | Curve Curve
                | Rect  Rectangle                       // changed
                | Wedge [AlgDrawable]

move :: Point NewAlgDrawable -> NewAlgDrawable
move p object
 = case object of
    Line  line         -> Line  (line + (p,p))
    Curve (rect,a1,a2) -> Curve (rect + (p,p),a1,a2)
    Rect  (p1,p2)      -> Rect  (p1+p,p2+p)             // changed
    Wedge parts        -> Wedge (map (move p) parts)

draw :: NewAlgDrawable -> Picture -> Picture
draw object
 = case object of
    Line  line  -> DrawLine line
    Curve curve -> DrawCurve curve
    Rect  rect  -> DrawRectangle rect                   // changed
    Wedge parts -> seq (map draw parts)
```

On the other-hand, adding an entirely new manipulation function is easier for the algebraic data type. Only the new function has to be defined. In the object-oriented approach, each object creating function should be changed accordingly.

As example we will add an operation that determines the bounding rectangle of drawable objects. For the algebraic data type approach we define the function

```
bounds :: AlgDrawable -> Rectangle
bounds object
 = case object of
    Line  line  -> normalize line
    Curve curve -> curveBounds curve
    Rect  rect  -> normalize rect
    Wedge parts -> foldl1 combine_bounds (map bounds parts)
```

using

```
foldl1 :: (a a -> a) [a] -> a
foldl1 f [x:xs] = foldl f x xs

combine_bounds :: Rectangle Rectangle -> Rectangle
combine_bounds ((tl1x,tl1y),(br1x,br1y)) ((tl2x,tl2y),(br2x,br2y))
 = ((min tl1x tl2x,min tl1y tl2y),(max br1x br2x,max br1y br2y))

normalize :: Rectangle -> Rectangle
normalize ((x1,y1),(x2,y2)) = ((min x1 x2, min y1 y2), (max x1 x2, max y1 y2))
```

For the object oriented approach we have to change the definition of `Drawable` and all functions generating objects of this type. We use the same supporting functions as above.

```
::  Drawable E.a =  { state  :: a
                    , move   :: Point -> a -> a
                    , draw   :: a Picture -> Picture
                    , bounds :: a -> Rectangle                          // new
                    }

MakeLine :: Line -> Drawable Void
MakeLine line
 =  { state  = line
    , move   = \dist line -> line + (dist,dist)
    , draw   = DrawLine
    , bounds = \l -> normalize l                                        // new
    }

MakeCurve :: Curve -> Drawable Void
MakeCurve curve
 =  { state  = curve
    , move   = \dist (rect,a1,a2) -> (rect + (dist,dist),a1,a2)
    , draw   = DrawCurve
    , bounds = \c -> curveBounds c                                      // new
    }

MakeRectangle :: Rectangle -> Drawable Void
MakeRectangle ((x1,y1),(x2,y2))
 =  { state  = [ MakeLine ((x1,y1),(x1,y2)), MakeLine ((x1,y2),(x2,y2))
               , MakeLine ((x2,y2),(x2,y1)), MakeLine ((x2,y1),(x1,y1))
               ]
    , draw   = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
    , move   = \d -> map (MoveDrawable d)
    , bounds = \parts -> foldl1 combine_bounds                          // new
                           (map (\{state,bounds} -> bounds state) parts)
    }

MakeWedge :: Curve -> Drawable Void
MakeWedge curve=:((tl,br), a1, a2)
 =  { state  = [ MakeLine (mid,mid+ epc1)
               , MakeLine (mid,mid+ epc2)
               , MakeCurve curve
               ]
    , draw   = \s p -> foldl (\pict {state,draw} -> draw state pict) p s
    , move   = \d -> map (MoveDrawable d)
    , bounds = \parts -> foldl1 combine_bounds                          // new
                           (map (\{state,bounds} -> bounds state) parts)
    }
where   mid = (tl + br)/ (2,2)
        (epc1,epc2) = EndPointsOfCurve curve
```

It is not possible to give a general rule to use either the object-oriented approach or the algebraic data type approach. As these examples show both approaches have their advantages and disadvantages. The decision should be based on the expected use and changes of the data type. Fortunately, usually neither of the choices is really wrong. It is commonly only a matter of convenience.

A pipeline of functions

Existentially quantified data structures can also be used as a solution to the following problem. Consider the function `seq` which applies a sequence of functions to a given argument (see also Chapter 5).

```
seq :: [t->t] t -> t
seq []  s = s
seq [f:fs] s = seq fs (f s)
```

Since all elements of a list must have the same type, only (very) limited sequences of functions can be composed with `seq`. In general it is not possible to replace `f o g o h` by `seq [h, g, f]`. The types of the argument and the final result as well as the types of all intermediate results might all be different. Applying the `seq` function forces all those types to become the same.

Existential types make it possible to hide the actual types of all intermediate results, as shown by the following type definition.

```
:: Pipe E.via a b   = Direct (a->b)
                    | Indirect (a->via) (Pipe Void via b)
```

Using this `Pipe` data type, it is possible to compose arbitrary functions in a real pipe-line fashion. The only restriction is that types of two consecutive functions should match. The function `ApplyPipe` for applying a sequence of functions to some initial value is defined as follows.

```
ApplyPipe:: (Pipe Void a b) a -> b
ApplyPipe (Direct f)        x   = f x
ApplyPipe (Indirect f pipe) x   = ApplyPipe pipe (f x)
```

The expression `ApplyPipe (Indirect toReal (Indirect exp (Direct toInt))) 7` is valid, and is typed with `Int`. The result is `1097`.

4.3 Uniqueness types

A very important property for reasoning about and analysing functional programs is *referential transparency*: functions always return the same result when called with the same arguments. Referential transparency makes it possible to reason about the evaluation of a program by substituting an application of a function with arguments by the functions definition in which for each argument in the definition uniformly the corresponding argument of the application is substituted. This principle of *uniform substitution*, which is familiar from high school mathematics, is vital in reasoning about functional programs.

Imperative languages like C, C++ and Pascal allow data to be updated destructively. This feature is not only important for reasons of efficiency (the memory reserved for the data is re-used again). The possibility to destructively overwrite data is a key concept on any computer. E.g. one very much would like to change a record stored in a database or the contents of a file. Without this possibility no serious program can be written. Incorporating destructive updating without violating referential transparency property of a functional program takes some effort.

The price we have to pay in imperative languages is that there is no referential transparency; the value of a function application can be dependent on the effects of the program parts evaluated previously. Uniqueness types are a possibility to combine referential transparency and destructive updates.

4.3.1 Graph Reduction

Until now we have not been very precise about the model of computation used in the functional language Clean. Since the number of references to an expression is important to determine whether it is unique or not, we must become a little bit more specific.

The basic ingredients of execution, also called *reduction*, have been discussed. The first principle we have seen is *uniform substitution*: an application of a function with arguments is replaced by the functions definition in which for each argument in the definition uniformly the corresponding argument of the application is substituted. The second principle is *lazy evaluation*: an expression is only evaluated when its value is needed.

Now we add the principle of *graph reduction*: all occurrences of a variable are replaced by one and the same expression during uniform substitution. The variables are either formal function arguments or expressions introduced as local definition. Clearly locally defined functions cannot be shared in the same way. This implies that expressions are never copied and hence an expression is evaluated at most once. The reason that it is safe to share expressions is the property called *referential transparency*: the value of an expression is independed of the context in which it is evaluated.

Graph reduction is illustrated by the following examples. A reductions step is indicated by the symbol →, a sequence of reduction steps is indicated by → . Whenever we find it useful we underline the *redex* (*red*ucable *ex*pression): the expression to be rewritten. Local definitions are used to indicate sharing.

```
Start = 3*7 + 3*7

Start
→   3*7 + 3*7
→   3*7 + 21
→   21 + 21
→   42
```

```
Start = x + x where x = 3*7

Start
→   x + x where x = 3*7
→   x + x where x = 21
→   42
```

Note that the sharing introduced in the rightmost program saves some work. These reduction sequences can be depicted as:

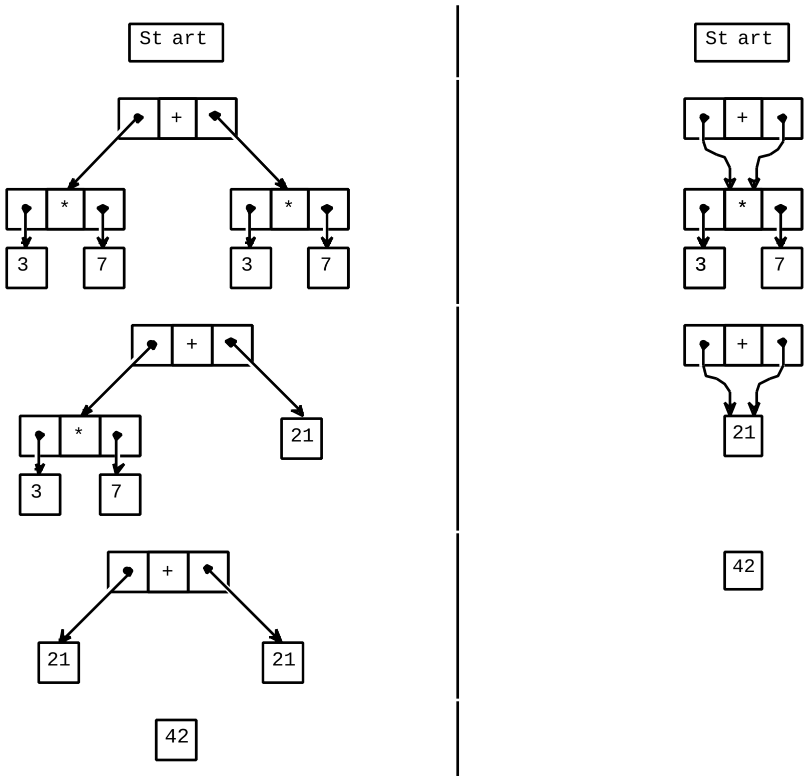

Figure 4.1: Pictorial representation of the reduction sequences shown above.

An other example where some work is shared is the familiar power function.

```
power :: Int Int -> Int
power x 0 = 1
power x n = x * power x (n-1)

Start :: Int
Start = power (3+4) 2
```

This program is executed by the following sequence of reduction steps.

```
Start
→   power (3+4) 2
→   x * power x (2-1) where x = 3+4
→   x * power x 1 where x = 3+4
→   x * x * power x (1-1) where x = 3+4
→   x * x * power x 0 where x = 3+4
→   x * x * 1 where x = 3+4
→   x * x * 1 where x = 7
→   x * 7 where x = 7
→   49
```

The number of references to an expression is usually called the *reference count*.

4.3.2 Destructive updating

Consider the special data structure which represents a file. This data structure is special since it represents a structure on a disk which (usually) has to be destructively updated. So, a program manipulating such a data structure is not only manipulating a structure inside the program but it is also manipulating a structure in the outside world. The Clean run-time system takes care of keeping the real world object and the structure inside your program up to date. In your program you just manipulate the data structure.

Assume that one would have a function `fwritec` that appends a character to an existing file independent of the context from which it is called and returns the modified file. So, the intended result of such a function would be a file with the extra character in it:

```
fwritec :: Char File -> File
```

Such a function could be used in a context by other functions:

```
AppendA :: File -> File
AppendA file = fwritec 'a' file
```

Let us now suppose that the following function `AppendAB` could be defined in a functional language.

```
AppendAB :: File -> (File, File)
AppendAB file = (fileA, fileB)
where   fileA = fwritec 'a' file
        fileB = fwritec 'b' file
```

What should then be the contents of the files in the resulting tuple `(fileA, fileB)`? There seem to be only two solutions, which both have unwanted properties.

The first is to assume that `fwritec` destructively changes the original file by appending a character to it (like in imperative languages). Then, the resulting tuple of `AppendAB` will depend on the evaluation order. If `fileB` is evaluated before `fileA` then `'b'` is appended to the file before `'a'`. If `fileA` is evaluated before `fileB` then the `'a'` will be written before `'b'`. This violates the rule of referential transparency in functional programming languages. So, just overwriting the file is rejected since loosing referential transparency would tremendously complicate analysing and reasoning about a program.

The second solution would be that in conformity with referential transparency the result is a tuple with two files: one extended with a character `'a'` and the other with the character `'b'`. This does not violate referential transparency because the result of the function calls `AppendA file` and `AppendB file` are not influenced by the context. This means that each function call `fwritec` would have to be applied on a 'clean' file, which in turn would mean that for the function call `AppendAB` two copies of the file have to be made. To the first copy the character `'a'` is appended, and to the second copy the character `'b'` is appended. If the original of the file is not used in any other context, it can be thrown away as garbage. This second solution however, does not correspond to the way operating systems behave in practice. It is rather impractical. This becomes even more obvious when one wants to write to a window on a screen: one would like to be able to perform output in an existing window. Following this second solution one would be obliged to construct a new window with each outputting command.

We cannot be satisfied with both solutions. We require that the result of any expression is well defined *and* we want to update files and windows without making unnecessary copies. So, what we need is a way to destructively update a data structure without violating referential transparency.

4.3.3 Environment passing

The way to deal with this problem may be typical for the way language designers think: "If you don't like it, don't allow it". In this situation we will not allow the update of a data structure representing real world objects when we must make copies in order to make the result properly defined.

The definition of `AppendAB` above should not be allowed in the language and therefore rejected by the compiler. This makes it possible for `fwritec` to destructively update the file. Semantically, one could say that it produces a new file while there is no reference anymore to the old file. So, as a kind of optimisation the old file can be reused to produce the result.

The problem is now moved to finding an alternative way to describe the wanted behaviour. Consider:

```
WriteAB :: File -> File
WriteAB file = fileAB
where   fileA  = fwritec 'a' file
        fileAB = fwritec 'b' fileA
```

Here, the data dependency is used which determine the order in which the characters are appended to the file (first `'a'`, then `'b'`)[4]. This programming style is very similar to the classical imperative style, in which the characters are appended by sequential program statements. Note however that the file to which the characters are appended is explicitly passed as an argument from one local function definition to another. This is very similar to adding two elements to a list.

The technique introduced is called environment passing: the argument is updated by the function and the result is passed as argument to another function. This is a very simple example of this technique. We will use more complicated examples in the rest of this chapter. Function that use this technique are also called *state transition* functions since the environment which is passed can be seen as a state which may be changed while it is passed.

4.3.4 Uniqueness information

Of course, somehow it must be determined (and specified) that the environment is passed properly i.e. in such a way that the required updates are possible. For this purpose a type system is designed which derives so-called *uniqueness properties.* A function is said to have an argument of *unique type* if there will be just a single reference to the argument

[4]The function `WriteAB` could have been defined alternatively as:

```
WriteAB:: File -> File
WriteAB file = fwritec 'b' (fwritec 'a' file)
```

Also here the `file` is actually passed as an environment (which is maybe a bit more complicated to see).

when the function will be evaluated. This property makes it safe for the function to re-use the memory consumed by the argument when appropriate.

In figure 4.1 all parts of the example of the left-hand side are unique. On the right-hand side the expression `3*7` is not unique since it is shared by both arguments of the addition. By drawing some pictures, it is immediatly clear that the function `WriteAB` introduced above uses the file unique, while in `AppendAB` the reference count of the file is 2.

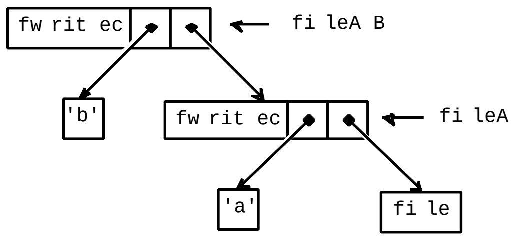

Figure 4.2: The result of `WriteAB file`

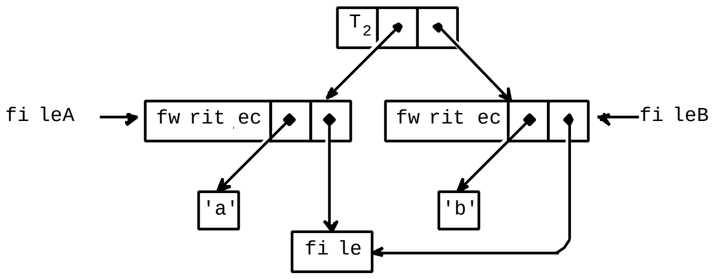

Figure 4.3: The result of `AppendAB file`

The function `fwritec` demands its second argument, the file, to be of unique type (in order to be able to overwrite it) and consequently it is derived that `WriteAB` must have a unique argument too. In the type this uniqueness is expressed with an asterisk which is attached as an attribute to the conventional type:

```
fwritec :: Char *File -> *File
WriteAB :: *File -> *File
```

The uniqueness type system is an extension on top of the conventional type system. When in the type specification of a function an argument is attributed with the type attribute unique (*) it is guaranteed by the type system that, upon evaluation of the function, the function has private ("unique") access to this particular argument.

The correctness of the uniqueness type is checked by the compiler, like all other type information except strictness information. Assume now that the programmer has defined the function `AppendAB` as follows:

```
AppendAB :: File -> (File, File)
AppendAB file = (fileA, fileB)
where   fileA = fwritec 'a' file
        fileB = fwritec 'b' file
```

The compiler will reject this definition with the error message:

```
<conflicting uniqueness information due to argument 2 of fwritec>
```

This rejection of the definition is caused by the non-unique use of the argument file in the two local definitions (assuming the type `fwritec :: Char *File -> *File`).

It is important to know that there can be many references to the object before this specific access takes place. For instance, the following function definition will be approved by the type system, although there are two references to the argument file in the definition. When `fwritec` is called, however, the reference is unique.

```
AppendAorB :: Bool *File -> *File
AppendAorB cond file
    | cond      = fwritec 'a' file
    | otherwise = fwritec 'b' file
```

So, the concept of *uniqueness typing* can be used as a way to specify locality requirements of functions on their arguments: If an argument type of a function, say `F`, is *unique* then in any concrete application of `F` the actual argument will have reference count 1, so `F` has indeed 'private access' to it. This can be used for defining (inherent) destructive operations like the function `fwritec` for writing a character to a file.

Observe that uniqueness of result types can also be specified, allowing the result of an `fwrite` application to be passed to, for instance, another call of `fwrite`. Such a combination of uniqueness typing and explicit environment passing will guarantee that at any moment during evaluation the actual file has reference count 1, so all updates of the file can safely be done in-place. If no uniqueness attributes are specified for an argument type (e.g. the `Char` argument of `fwritec`), the reference count of the corresponding actual argument is generally unknown at run-time. Hence, no assumption can be made on the locality of that argument: it is considered as *non-unique*.

Offering a unique argument if a function requires a non-unique one is safe. More technically, we say that a unique object can be *coerced* to a non-unique one. Assume, for instance, that the functions `freadc` and `fwrites` have type

```
freadc:: File -> (Bool, Char, File)      // The Boolean indicates success or failure
fwrites :: String *File -> *File.
```

in the application

```
readwrite :: String *File -> (Bool, Char, File)
readwrite s f = freadc (fwrites s f)
```

the (unique) result `File` of `fwrites` is coerced to a non-unique one before it is passed to `freadc`.

Of course, offering a non-unique object if a function requires a unique one always fails. For, the non-unique object is possible shared, making a destructive update not well-defined. Note that an object may lose its uniqueness not only because uniqueness is not required by the context, but also because of sharing. This, for example, means that although an application of `fwritec` itself is always unique (due to its unique result type), it is considered as non-unique if there exist more references to it. To sum up, the *offered* type (by an argument) is determined by the *result* type of its outermost application and the reference count of that argument.

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 object type itself). As we have seen in the example above the reference count is computed for each right-hand side separatly. When there is an expression in the gaurds requiring an unique object this must be taken into account. This is the reason we have to write:

```
AppendAorB:: *File -> *File
AppendAorB file
    | fc == 'a' = fwritec 'a' nf
    | otherwise = fwritec 'b' nf
where
    (_,fc,nf)   = freadc file
```

When the right-hand side of `AppendAorB` is evaluated, the guard is determined first (so the resulting access from `freadc` to `file` is not unique), and subsequently one of the alternatives is chosen and evaluated. Depending on the condition `fc == 'a'`, either the reference from the first `fwritec` application to `file` or that of the second application is left and therefore unique. As you might expect it is not allowed to use `file` instead of `nf` in the right-hand sides of the function `AppendAorB`. File manipulation is explained in more detail in chapter 5.

4.3.5 Propagation of uniqueness

Pattern matching is an essential aspect of functional programming languages, causing a function to have access to 'deeper' arguments via 'data paths' instead of via a single reference. For example, the head function for lists, which is defined as

```
head :: [a] -> a
head [hd:tl] = hd
```

has (indirect) access to both hd and tl. This deeper access gives rise to, what can be called, indirect sharing: several functions access the same object (via intermediate data constructors) in spite of the fact that the object itself has reference count 1. Consider, for example the function heads which is defined as follows.

```
heads list = (head list, head list)
```

In the right-hand side of heads, both applications of head retrieve their result via the same list constructor.

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 could say that uniqueness of list elements *propagates outwards*: if a list contains unique elements, the list itself should be unique as well. One can easily see that, without this propagation requirement, locality of object cannot be guaranteed anymore. E.g., suppose we would admit the following type for head.

```
head :: [*a] -> *a
```

Then, the definition of heads is typable, for, there are no uniqueness requirements on the direct argument of the two head applications. The type of heads is:

```
heads :: [*a] -> (*a,*a)
```

which is obviously not correct because the same object is delivered twice. However, applying the uniqueness propagation rule leads to the type

```
head :: *[*a] -> *a
```

Indeed, this excludes sharing of the list argument in any application of head, and therefore the definition of heads is no longer valid.

In general, the propagation rule reflects the idea that if an unique object is stored in a larger data structure, the latter should be unique as well. This can also be formulated like: an object stored in a data structure can only be unique when the data structure is unique. When we have the constraint that an element of a data structure is unique, this implies that also the data structure is unique.

In practice, however, one can be more liberal when the evaluation order is taken into account. The idea is that multiple references to an (unique) object are harmless if one knows that only one of the references will be present at the moment the object is accessed destructively. For instance, the compiler 'knows' that only one branch of the predefined conditional function if will be used. This implies that the following function is correct.

```
transform :: (Int -> Int) *{#Int} -> *{#Int}
transform f s
 | size s == 0 = s
 | otherwise   = if (s.[0] == 0)
                   {f i \\ i <-: s}
                   {f i \\ _ <-: s & i <- [s.[0]..]}
```

This example shows also that strictness of objects is not restricted to files and windows.

4.3.6 Uniqueness polymorphism

To indicate that functions leave uniqueness properties of arguments unchanged, one can use (*uniqueness*) *attribute variables*. The most simple example is the identity function which can be typed as follows:

```
id :: u:a -> u:a
```

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. Note that we tacitly assume an attribute for 'non-unique' although there exists no denotation for it in Clean.

A more interesting example is the function `freadc`[5] which is typed as

```
freadc :: u:File -> (Bool, 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 as well as for writing. In the second case the resulting file is also not unique, hence write access is not permitted.

One can also indicate relations between attribute variables appearing in the type specification of a function, by adding so called *coercion statements*. These statements consist of attribute 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 operator `++` is shown.

```
(++) infixr 5 :: v:[u:a] w:[u:a] -> x:[u:a],        [v w x<=u, w<=x]
```

The first coercion statement express 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 latter statement `w<=x` expresses that the spine uniqueness[6] of `append`'s result depends only on the spine attribute `w` of the second argument. This reflects the operational behaviour of `append`, namely, to obtain the result list, the first list argument is fully copied, after which the final tail pointer is redirected to the second list argument.

```
(++) [hd:tl] list = [hd: tl ++ list]
(++) _       list = list
```

In Clean it is permitted to omit attribute variables and attribute inequalities that arise from propagation properties; those will be added automatically by the type system. As a consequence, the following type specification for `++` is also valid.

```
(++) infixr 5 :: [u:a] w:[u:a] -> x:[u:a],      [w<=x]
```

Of course, it is always allowed to use a more specific type (by instantiating type and/or attribute variables). All types given below are valid types for `++`.

```
(++) infixr 5 ::  [u:a] x:[u:a]  -> x:[u:a]
(++) infixr 5 :: *[*Int] *[*Int] ->  *[*Int]
(++) infixr 5 ::  [a]    *[a]    ->  *[a]
```

To make types more readable, Clean offers the possibility to use *anonymous* attribute variables as a shorthand for attribute variables of which the actual names are not essential. Using anonymous attributes `++` can be typed as follows.

```
(++) infixr 5 :: [.a] v:[.a] -> w:[.a],     [v<=w]
```

This is the type derived by the compiler. 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.

[5]The function `freadc` defined in the 0.8 library is not as general as the function shown here. In the 0.8 library there are separate functions to read from a unique file and from an shared file.

[6]The spine of a list is the structure of Cons nodes connecting the elements.

Finally, we can always use an unique list where an ordinary list is expected. So, it is sufficient to specify the following type for append:

```
(++) infixr 5::[.a] u:[.a] -> u:[.a]
```

Apart from strictness annotations this is the type specified for the append operator in the module `StdList` of the standard environment.

4.3.7 Attributed data types

First, remember that 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, a programmer does not have change the corresponding data type definition itself; the type system of Clean will generate appropriate uniqueness variants for the (classical) types of all data constructors. Such a uniqueness variant is obtained via a consistent attribution of all types and subtypes appearing in a data type definition. E.g., for `Cons` this attribution yields the type

```
Cons :: u:a v:(List u:a) -> v:List u:a, [v<=u]
```

Describing the attribution mechanism in all its details is beyond the scope of this book; the procedure can be found in the reference manual and [Barendsen 93]. The main property is that all type variables and all occurrences of the defined type constructor, say T, will receive an attribute variable. Again this is done in a uniform way: equal variables will receive equal attributes, and the occurrences of T are equally attributed as well. Besides that, attribute variables are added at non-variable positions if they are required by the propagation properties of the corresponding type constructors (see example below). The coercion statements are, as usual, determined by the propagation rule. As an example, consider the following `Tree` definition.

```
:: Tree a = Node a [Tree a]
```

The type of the data constructor `Node` is

```
Node :: u:a w:[v:Tree u:a] -> v:Tree u:a,   [v<=u, w<=v]
```

Observe that the uniqueness variable `w` and the coercion statement `[w<=v]` are added since the list type ‘propagates’ the `v` attribute of its element.

One can also specify that a part of data type definition, should receive the same attribute as the defined type constructor, say `T`, itself. For this purpose the anonymous (.) attribute variable is reserved, which can be attached to any (sub) type on the right-hand of `T`’s definition. The idea is that the dot attribute denotes the same attribute as the one assigned to the occurrences of `T`. This is particularly useful if one wants to store functions into data structures; see also the next section on higher order uniqueness typing. For example, the following definition of `Tree`

```
:: Tree2 = Node2 .Int [Tree2]
```

causes the type for the data constructor `Node` to be expanded to

```
Node2 :: u:Int v:[u:Tree2] -> u:Tree2,      [v<=u]
```

Unfortunately, the type attribute `w` is not used in the result of the constructor `Node`. Hence, there is no way to store the uniqueness of the arguments of `Node` in its type. So, in contrast with the type `List`, it is not possible to construct unique instances of the type `Tree`. This implies that the function to reverse[7] trees

```
swap (Node a leafs) = Node a [swap leaf \\ leaf <- rev leafs]

rev :: ![.a] -> [.a]
rev l = rev_ l []
where   rev_ [a:x] l = rev_ x [a:l]
        rev_ []    l = l
```

obtains type

```
swap :: !(Tree a) -> Tree a;
```

instead of

```
swap :: !u:(Tree .a) -> u:(Tree .a)
```

This implies that an unique tree will loose its uniqueness attribute when it is swapped by this function `swap`. Due to the abbreviations introduced above the last type can also be written as:

```
swap :: !(Tree .a) -> (Tree .a)
```

When we do need unique instances of type `Tree`, we have to indicate that the list of trees inside a node has the same uniqueness type attribute as the entire node:

```
:: Tree a = Node a .[Tree a]
```

Now the compiler will derive and accept the type that indicates that swapping an unique tree yields an unique tree: `swap :: !(Tree .a) -> (Tree .a)`.

When all `Trees` ought to be unique we should define

```
:: *Tree a = Node a [Tree a]
```

The corresponding type of the function `swap` is

```
swap :: !*(Tree .a) -> *Tree .a;
```

In practice the pre-defined attribution scheme appears to be too restrictive. First of all, it is convenient if it is allowed to indicate that certain parts of a data structure are always unique. Take, for instance, the type `Progstate`, defined in chapter 5 containing the (unique) file system of type `Files`.

```
:: *ProgState   = {files :: Files}
```

According to the above attribution mechanism, the `Files` would have been non-unique. To circumvent this problem, one can make `Progstate` polymorphic, that is to say, the definition `Progstate` becomes

```
:: Progstate file_system = {files :: file_system}
```

Then, one replaces all uses of `Progstate` by `Progstate *Files`. This is, of course, a bit laborious, therefore, it is permitted to include `*` attributes in data type definitions themselves. So, the definition of `Progstate`, is indeed valid. Note that, due to the outwards propagation of the `*` attribute, `Progstate` itself becomes unique. This explains the `*` on the left-hand side of the definition of `Progstate`.

4.3.8 Higher order uniqueness typing

Higher-order functions give rise to partial (often called *Curried*) applications (of functions as well as of data constructors), i.e. applications in which the actual number of arguments is less than the arity of the corresponding symbol. If these partial applications con-

[7]The function `reverse` provided in `StdList` of Clean 1.1 and 1.2 has a type that is to restrictive. It is not able to reverse an unique list of unique elements to an unique list of unique objects:

```
reverse::!.[a] -> [a]
reverse list = reverse_ list []
where   reverse_::![a] [a] -> [a]
        reverse_ [hd:tl] list   = reverse_ tl [hd:list]
        reverse_ [] list        = list
```

tain 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, we will illustrate that it is dangerous to allow the above application to be shared. For example, if `(fwritec unifile)` is passed to a function

```
WriteAB write_fun = (write_fun 'a', write_fun 'b')
```

Then the argument of `fwritec` is not longer unique at the moment one of the two write operations takes 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, the uniqueness type system 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 actual 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. This is exactly what we want for files.

To define data structures containing Curried applications it is often convenient to use the (anonymous) dot attribute. Example

```
:: Object a b = { state :: a
                , fun :: .(b -> a)
                }

new :: * Object *File Char
new = {state = unifile, fun = fwritec unifile}
```

Since `new` contains an essentially unique expression it becomes essentially unique itself. So, the result of `new` can only be coerced to a unique object implying that in any containing `new`, the attribute type requested by the context has to be unique.

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:([.] -> 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 consisting of the concatenation of `other_list` and `some_list`. Let us 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` becomes `*` as well. 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. The general rule for determining the uniqueness type of Curried variants of (function or data) symbols can be found in the reference manual and [Barendsen 93].

4.3.9 Creating unique objects

In the preceding subsections we showed how unique objects can be manipulated. The questions that remains is how to become the initial unique object. All unique objects corresponding with real world entities, like files and windows, are retrieved from the `world`. This is explained in detail in chapter I.5.

It is also possible to have unique objects of other types. Especially for arrays it is useful to have unique instances, since only unique arrays can be updated. There is a function `createArray` in `StdEnv` that can be used to create unique arrays. Array comprehensions can also be used to create unique arrays. Furthermore, all objects with reference count one created by unique function applications are unique. The `Start` expression itself is unique. By careful design of functions, this uniqueness can be passed to the places where it is needed.

4.3.10 Combining uniqueness typing and type classes

This is complicated stuff. Either to be omitted or to be written.

4.4 Exercises

4.1 Define an instance for type `Q` of the standard class `Arith`.

```
class Arith a | PlusMin, MultDiv, abs, sign, ~ a
```

4.2 Define complex numbers similar to `Q` and specify an instance of the class `Arith` for this new type.

4.3 Define an instance of `PlusMin` for lists `[a]` such that, for instance, the addition of two lists takes place element wise (if necessary, the shortest list is extended with zeros to obtain two lists of equal length). So, `[1,2,3] + [4,5] = [5,7,3]`.

4.4 Why should a `Pipe` object contain at least one function (each `Pipe` object ends with a `Direct` constructor containing the final function to be applied)? One can image a definition of `Pipe` with a kind of Nil constructor with no argument as a terminator.