Part I
Chapter 6
Advanced Programming

In this chapter we discus a number of advanced topics concerning writing functional programs. You should not study this chapter before you are able to work with the concepts introduced in the previous chapters. In fact you can switch to part II now and return to this chapter later on.

In section 1 we show how you can reason about the efficiency of programs and how you can increase the speed of programs. In the second section we demonstrate how the equivalence of functions definitions can be shown. Next we show how you can trace the execution of programs. The fourth section treats higher order functions on lists. Since lists are a heavily used data structure in functional programming languages, some general purpose list processing functions have been defined. Many list manipulations can be expressed in terms of these general purpose functions. The advantage of using these toolbox functions is that they can make the structure of the list processing clear and they facilitates some transformations. Moreover, they offer a very compact notation.

6.1 Efficiency of programs

Until now we haven't bothered much about the efficiency of the programs we have written. We think this is the way it should be. Correctness and clearness is in general more important than speed. However, sooner or later you will create a program that occurs to be (too) slow. In this section we provide you with the necessary tools to understand the efficiency of your programs.

There are two important aspects of efficiency that deserve some attention. The first aspect is the amount of time needed to execute a given program. This time complexity is discussed in some detail in this chapter. The other aspect is the amount of memory space needed to compute the result. Reasoning about the space efficiency of programs in a lazy language is somewhat more complicated in a lazy functional language. Although we give some hints on space efficiency in this chapter, we delay the thorough discussion to part III. Especially programs manipulating strict values of basic types perform often much better as you might expect using the notions of this chapter.

In order to understand the time efficiency of programs we first argue that counting the number of reduction steps in generally a better measure than counting bare seconds. Next we show how we usually work more easily with a proper approximation of the number of reduction steps. Furthermore, we give some hints how to improve the efficiency of programs.

6.1.1 The unit to measure efficiency

When you have to measure time complexity of your program you first have to decide which units will be used. Perhaps your first idea is to measure the execution time of the program in seconds. There are two problems with this approach. The first problem is that the execution time is dependent of the actual machine used to execute the program. The second problem is that the execution time is generally dependent on the input of the program. Also the implementation of the programming language used has generally on important influence. Especially in the situation that there are interpreters and compilers involved or implementations from various manufactures.

In order to overcome the first problem we measure the execution time in reduction steps instead of in seconds. Usually it is sufficient to have an approximation of the exact number of reduction steps. The second problem is handled by specifying the number of reduction steps as function of the input of the program. This is often called the *complexity* of the program.

For similar reasons we will use nodes to measure the space complexity of a program or function. The space complexity is also expressed as function of the input. In fact we distinguish the total amount of nodes used during the computation and the maximum number of nodes used at the same moment to hold an (intermediate) expression. Usually we refer to the maximum number of nodes needed at one moment in the computation as the space complexity.

6.1.2 Complexity

The time complexity of a program (or function) is an approximation of the number of reduction steps needed to execute that program. The space complexity is an approximation of the amount of space needed for the execution. It is more common to consider time complexity than space complexity. When it is clear from the context which complexity is meant we often speak of the complexity.

Upper bounds

We use the *O*-notation to indicate the approximation used in the complexity analysis. The *O*-notation gives an upper bound of the number of reductions steps for sufficient large input values. The expression $O(g)$ is pronounced as *big-oh* of *g*. Formally this is defined as:

> Let f and g be functions. The notation $f(n)$ is $O(g(n))$ means that there are positive numbers c and m such that for all arguments $n \geq m$ we have $|f(n)| \leq c \times |g(n)|$. So, $c \times |g(n)|$ is an upper bound of $|f(n)|$ for sufficient large arguments.

We usually write $f(n) = O(g(n))$, but this can cause some confusion. The equality is not symmetric; $f(n) = O(g(n))$ does not imply $O(g(n)) = f(n)$. This equality is also not transitive; although $3\times n^2 = O(n^2)$ and $7\times n^2 = O(n^2)$ this does not imply that $3\times n^2 = 7\times n^2$. Although this is a strange equality we will use it frequently.

As example we consider the function $f(n) = n^2 + 3 \times n + 4$. For $n \geq 1$ we have $n^2 + 3 \times n + 4 \leq n^2 + 3 \times n \times n + 4 \times n^2 = n^2 + 3 \times n^2 + 4 \times n^2 = 8 \times n^2$. So, $f(n) = O(n^2)$.

Keep in mind that the *O*-notation provides an upper bound. There are many upper bounds for a given function. We have seen that $3\times n^2 = O(n^2)$, we can also state that $3\times n^2 = O(n^3)$, or $3\times n^2 = O(2^n)$.

We can order functions by how fast their value grows. We define $f < g$ as $f = O(g)$ and $g \neq O(f)$. This means that g grows faster than f. In other words:

$$f < g \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

The notation $f \equiv g$ indicates that the functions f and g grow equally fast: $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Using this notation we can list the order of the most occurring complexity functions:

$1 = n^0 < \log(\log n) < \log n < n^b < n = n^1 < n^c < n^{\log n} < c^n < n^n < c^{c^n}$,
where $0 < b < 1 < c$.

In addition we have an ordering within functions of a similar form:

$n^b < n^c$ when $0 < b < c$;

$c^n < d^n$ when $1 < c < d$.

In order to give you some idea how fast the number of reduction steps grows for various complexity functions we list some examples in the following table:

function	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$	name
$O(1)$	1	1	1	1	Constant
$O(^{10}\log n)$	1	2	3	4	Logarithmic[1]
$O(\sqrt{n})$	3	10	32	100	
$O(n)$	10	100	1000	10000	Linear
$O(n\ ^{10}\log n)$	10	200	3000	40000	
$O(n^2)$	100	10000	10^6	10^8	Quadratic
$O(n^3)$	1000	10^6	10^9	10^{12}	Cubic
$O(2^n)$	1024	10^{30}	10^{300}	10^{3000}	Exponential

Table 1: Number of reductions steps as function of the input and complexity.

Some of these numbers are really huge. Consider that the number of bits on a 1 GByte hard disk is 10^{10}, light travels about 10^{16} meters in one year, and the mass of our sun is about 2×10^{30} Kg. At a speed of one million reduction steps per second a program of 10^{30} reductions takes about 3×10^{16} = 30,000,000,000,000,000 years. As a reference, the age of the universe is estimated at 12×10^9 year. It is currently unlikely that your machine executes a functional program significantly faster than 10^6 reduction steps per second. Using the same reduction speed, 10^{12} reduction steps takes about twelve days, this is probably still more than you want for the average program.

Under bounds

In addition to the approximation of the upper bound of the number of reductions steps needed, we can give an approximation of the under bound of the amount of reductions needed. This is the number of reductions that is needed at least. We use the Ω-notation, pronounced omega notation, for under bounds. The Ω-notation is defined using the O-notation:

$$f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$$

This implies that it is a similar approximation as the upper bound. It is only valid for large arguments and constants are irrelevant.

As an example we consider again the function $f(n) = n^2 + 3 \times n + 4$. An under bound for those function is the constant. $f(n) = \Omega(1)$. In fact this is an under bound of any function, we cannot expect anything to grow slower than no increase at all. With a little more ef-

[1] We used logarithms with base 10 in this table since we use powers of 10 as value for n. A logarithm with base 2 is more common in complexity analysis. This differs only a constant factor (2.3) in the value of the logarithm.

fort we can show that $f(n) = \Omega(n)$ and even $f(n) = \Omega(n^2)$. This last approximation can be justified as follows: for any $n \geq 0$ we have $f(n) = n^2 + 3 \times n + 4 > n^2 + 3 \times n \geq n^2$.

Tight upper bounds

As we have seen upper bounds and under bounds can be very rough approximations. We give hardly any information by saying that a function is $\Omega(1)$ and $O(2^n)$. When the upper bound and under bound are equal we have tight bounds around the function, only the constants in the asymptotic behaviour are to be determined. We use the Θ-notation, pronounced *theta notation*, to indicate tight upper bounds:

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \quad f(n) = \Omega(g(n))$$

For the function $f(n) = n^2 + 3 \times n + 4$ we have seen $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$. This makes it obvious that $f(n) = \Theta(n^2)$.

6.1.3 Counting reduction steps

Now we have developed the tools to express the complexity of functions. Our next task is to calculate the number of reduction steps required by some expression or function to determine the time complexity, or the number of nodes needed to characterise the space complexity. When there are no recursive functions (or operators) involved this is simply a matter of counting. All these functions will be of complexity $\Theta(1)$.

Our running example, $f(n) = n^2 + 3 \times n + 4$. has time complexity $\Theta(1)$. The value of the function itself grows quadratic, but the amount of steps needed to compute this value is constant: two multiplications and three additions. We assume that multiplication and addition is done in a single instruction of your computer, and hence in a single reduction step. The amount of time taken is independent of the value of the operands. The number of nodes needed is also constant: the space complexity is also $\Theta(1)$.

For recursive functions we have to look more carefully at the reduction process. Usually the number of reduction steps can be determined by an inductive reasoning. As example we consider the factorial function `fac`:

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

For any non-negative argument this takes `3*n+1` reduction steps (for each recursive call one for `fac`, one for `*` and one for `-`). Hence, the time complexity of this function is $\Theta(n)$. As a matter of fact also the space complexity is $\Theta(n)$. The size of the largest intermediate expression `n * (n-1) * … * 2 * 1` is proportional to `n`.

Our second example is the naive Fibonacci function:

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

Computing `fib n` invokes the computation of `fib (n-1)` and `fib (n-2)`. The computation of `fib (n-1)` on its turn also calls `fib (n-2)`. Within each call of `fib (n-2)` there will be two calls of `fib (n-4)`. In total there will be one call of `fib (n-1)`, two calls of `fib (n-2)`, three calls of `fib (n-3)`, four calls of `fib (n-4)`.... The time (and space) complexity of this function is greater than any power. Hence, `fib n` = $\Theta(2^n)$: the number of reduction steps grows exponentially.

memorisation

It is important to realise that the complexity is a property of the algorithm used, but not necessarily a property of the problem. For our Fibonacci example we can reduce the complexity to $O(n)$ when we manage to reuse the value of `fib (n-m)` when it is needed

again. Caching these values is called memorisation. A simple approach is to generate a list of Fibonacci numbers. The first two elements will have value `1`, the value of all other numbers can be obtained by adding the previous two.

```
fib2 :: Int -> Int
fib2 n = fibs!n

fibs :: [Int]
fibs =: [1,1:f n \\ n <- [2..]]
where   f n = fibs!(n-1) + fibs!(n-2)
```

Now we want to determine the time complexity of `fib2`. It is obvious that selecting element `n` from list `fibs` takes `n` reduction steps. This is $O(n)$. To compute element `n` of the list `fibs` we need to calculate element `n-1` (and hence all previous elements) and to do two list selections each of $O(n)$ and one addition. This makes the time complexity of `fibs` $O(n^2)$. The space required is proportional to the length of the list of Fibonacci numbers: $O(n)$.

It is not useful to make the list of Fibonacci numbers a global constant when only one Fibonacci number is needed in your program. Nevertheless it is essential to achieve the sharing that all numbers used as intermediate values are obtained form one and the same list. This can be achieved by making the `fibs` a local definition of the function `fib2`.

The generation of the list of Fibonacci numbers can be further improved. To compute the next Fibonacci number we need the previous two. Our current algorithm selects these numbers from the list as if this is a random selection. This selection takes $O(n)$ reduction steps. The following function to generate a list of Fibonacci function uses the known order in which the elements are used to compute the next element:

```
fibs2 :: [Int]
fibs2 =: [1,1:[a+b \\ a <- fibs2 & b <- tl fibs2]]
```

When you prefer explicit recursive functions instead of the list comprehension, this algorithm can be written as:

```
fibs3 :: [Int]
fibs3 =: [1,1:map2 (+) fibs3 (tl fibs3)]

map2 :: (a b->c) [a] [b] -> [c]
map2 f [a:x] [b:y] = [f a b:map2 f x y]
map2 _ _     _     = []
```

The second alternative of `map2` is never used in this application. It is only included to make `map2` a generally useful function.

Computing the next element of the list takes `fib3` takes only two reductions (`map2` and `+`). This makes this algorithm to compute Fibonacci numbers $O(n)$ in time and space.

When our program needs only a single Fibonacci number there is no need to create a list of these numbers as a CAF. Since only the two previous numbers are used to compute the next Fibonacci number, there is no need at all to store them in a list. This is used in the following function to compute Fibonacci numbers.

```
fib4 n = f n 1 1
where   f :: !Int !Int !Int -> Int
        f 0 a b = a
        f n a b = f (n-1) b (a+b)
```

Computing the next Fibonacci number takes three reduction steps. So, this algorithm has a time complexity of $O(n)$. By making the local function `f` strict in all of its arguments we achieved that these arguments are evaluated before `f` is evaluated. This makes the space required for the intermediate expressions a constant. The space complexity of this version of the Fibonacci function is $O(1)$.

The complexity analysis shows that the function `fib3` behaves much better than `fib`. Although the function `fib` is clearer, it is obvious that `fib3` is preferred in programs. List

selection using `fibs2` or `fibs3` has the same time complexity, but is only worthwhile when a multiple Fibonacci numbers are used.

Determining the complexity for recursive functions

In the examples above a somewhat ad hoc reasoning is used to determine the complexity of functions. In general it is a convenient to use a function indicating the number of reduction steps (or nodes) involved. Using the definition of the recursive function to analyse it is possible to derive a recursive expression for the complexity function $C(n)$. The complexity can than be settled by an inductive reasoning. The next table lists some possibilities (c and d are arbitrary constants ≥ 0):

$C(n)$	Complexity
$\frac{1}{2} \times C(n\text{-}1)$	$O(1)$
$\frac{1}{2} \times C(n\text{-}1) + c \times n + d$	$O(n)$
$C(n\text{-}1) + d$	$O(n)$
$C(n\text{-}1) + c \times n + d$	$O(n^2)$
$C(n\text{-}1) + c \times n^x + d$	$O(n^{x+1})$
$2 \times C(n\text{-}1) + d$	$O(2^n)$
$2 \times C(n\text{-}1) + c \times n + d$	$O(2^n)$
$C(n/2)$	$O(1)$
$C(n/2) + d$	$O(\log n)$
$C(n/2) + c \times n + d$	$O(n)$
$2 \times C(n/2) + d$	$O(n)$
$2 \times C(n/2) + c \times n + d$	$O(n \log n)$
$4 \times C(n/2) + c \times n^2 + d$	$O(n^2 \log n)$

Table 2: The complexity for a some recursive relations of the number of reduction steps $C(n)$.

Although this table is not exhaustive, it is a good starting point to determine the complexity of very many functions.

As example we will show how the given upperbound, $O(\log n)$, of the complexity function C can be verified for $C(n) = C(n/2)+d$. We assume that $C(n/2)$ is $O(\log n/2)$. This implies that there exists positive numbers a and b such that $C(n/2) \leq a \times \log n/2 + b$ for all $n \geq 2$.

$C(n)$
$= C(n/2)+d$ // Using the recursive equation for $C(n)$
$\leq a \times \log (n/2) + b + d$ // Using the induction hypothesis
$= a \times (\log n - \log 2) + b + d$ // $\log (x/y) = \log x - \log y$
$= a \times \log n + b + d - a$ // arithmetic
$\leq a \times \log n + b$ iff $d \leq a$

We are free to chose positve values a and b. So, we can take a value a such that $a \geq d$ for any d. When we add the fact that $C(0)$ can be done in some finite time, we have proven that $C(n) = O(\log n)$.

In exactly the same we we can show that $C(n) = C(n/2)+d$ implies that $C(n) = O(n)$. For our proof with induction we assume now that $C(n/2) \leq a \times n/2 + b$. The goal of our proof is that this implies also that $C(n) \leq a \times n + b$.

$C(n)$
$= C(n/2) + d$ // Using the recursive equation for $C(n)$
$\leq a \times n/2 + b + d$ // Using the induction hypothesis
$\leq a \times n + b + d$ // Since a and n are posirive

For the same reasons as above this implies that $C(n) = O(n)$. This is consistent with our claim that we only determine upperbounds and the ordering on functions: $\log n < n$.

When we would postulate that $C(n) = C(n/2)+d$ implies that $C(n) = O(1)$ we have as induction hypotesis $C(n/2) \leq b$.

$C(n)$
$= C(n/2) + d$ // Using the recursive equation for $C(n)$
$\leq b + d$ // Using the induction hypothesis

But $C(n) = O(1)$ this implies that $C(n) \leq b$. This yields a contradiction. For arbitray d the equation $b + d \leq b$ is *not* valid. $C(n) = C(n/2)+d$ only implies that $C(n) = O(1)$ when $d = 0$. This is a special case in table 2.

As illustration of the application of thses rules we return to the complexity of some of our examples. The number of reduction steps of the factorial example above we have $C(\texttt{n}) = C(\texttt{n-1}) + 3$, hence the complexity is $O(\texttt{n})$. For the naive Fibonacci function `fib` we have $C(\texttt{n}) = C(\texttt{n-1}) + C(\texttt{n-2}) + 4 \leq 2 \times C(n\text{-}1)$, this justifies our claim that this function has complexity $O(2^n)$. The time complexity to compute element `n` of the list `fibs` is $C(\texttt{n-1})$ to compute the preceding part of the list, plus two list selections of `n` and `n-1` reductions plus two subtractions and one addition. This implies that $C(\texttt{n}) \leq C(\texttt{n-1}) + 2\times\texttt{n} + 4$, so the complexity is indeed $O(n^2)$. For `fib3` we have $C(\texttt{n}) = C(\texttt{n-1}) + 3$. This implies that this function is $O(\texttt{n})$.

Manipulation recursive data structures

When we try to use the same techniques to determine the complexity of the naive `reverse` function we immediately run into problems. This function is defined as:

```
reverse :: [a] -> [a]
reverse []    = []
reverse [a:x] = reverse x ++ [a]
```

The problem is that the value of the argument is largely irrelevant. The length of the list determines the number of reduction steps needed, not the actual value of these elements. For a list of n elements we have $C(n)$ is equal to the amount of work to reverse a list of length n-1 and the amount of work to append `[a]` to the reversed tail of the list. Looking at the definition of the append operator it is obvious that this takes a number of steps proportional to the length of the first list: $O(n)$.

```
(++) infixr 5 :: ![a] [a] -> [a]
(++) [hd:tl] list = [hd:tl ++ list]
(++) nil     list = list
```

For the function `reverse` we have $C(\texttt{n}) \leq C(\texttt{n-1}) + \texttt{n} + 1$. Hence the function `reverse` is $O(n^2)$.

Again the complexity is a property of the algorithm used, not necessarily of property of the problem. It is obvious that we cannot reverse a list without processing each element in the list, this is $O(n)$. It is the application of the append operator that causes the complexity to grow to $O(n^2)$. Using an other definition with an additional argument to hold the part of the list reversed up to the current element accomplish an $O(\texttt{n})$ complexity.

```
reverse :: [a] -> [a]
reverse l = rev l []
where   rev [a:x] l = rev x [a:l]
        rev []    l = l
```

For this function we have $C(\texttt{n}) = C(\texttt{n-1}) + 1$. This implies that this function is indeed $O(n)$. We have argued that it is inevitable to process each element at least once in order to reverse a list. So, this is also an under bound.

Using such an additional argument to accumulate the result of the function appears to be useful in many situations. This kind of argument is called an *accumulator*.

The analysis of functions that behave differently based on the value of the list elements is somewhat more complicated. In chapter 3 we introduced the following definition for insertion sort.

```
isort :: [a] -> [a] | Ord a
isort []    = []
isort [a:x] = Insert a (isort x)

Insert :: a [a] -> [a] | Ord a
Insert e []     = [e]
Insert e [x:xs]
    | e<=x      = [e,x : xs]
    | otherwise = [x : Insert e xs]
```

It is obvious that this algorithm requires one reduction step of the function `isort` for each element of the list. The problem is do determine the number of reductions required by `insert`. This heavily depends on the values in the list used as actual argument. When this list happens to be sorted `e<=x` yields always `True`. Only a single reduction step is required for each insertion in this situation. In this best case, the number of reduction steps is proportional to the length of the list, the complexity is $O(n)$.

When the argument list is sorted in the inverse order and all arguments are different the number of reduction steps required by `Insert` is equal to the number of elements that are already sorted. Hence, the number of reduction steps is `n` for the function `isort` and `1 + 2 + … + (n-1) + n` steps for inserting the next element in the sorted sub-list. From mathematics it is known that

$$1 + 2 + \ldots + (n-1) + n = \sum_{1}^{n} i = \frac{n\,(n-1)}{2}$$

The total number of reduction steps needed is `n + n*(n-1)/2`. This shows that the worst case complexity of the insertion sort algorithm is $O(n^2)$.

Note that we used a different technique to compute the complexity here. We computed the number of reduction steps required directly. It is often hard to compute the number reductions required directly. Using the recursive function $C(n)$, we have $C(n) = C(n-1) + n$. This implies that the complexity determined in this way is also $O(n^2)$.

The amount of reduction steps needed to sort an arbitrary list will be somewhere between the best case and the worst case. On average we expect that a new element will be inserted in the middle of the list of sorted elements. This requires half of the reduction steps of the worst case behaviour. This makes the expected number of reduction steps for an average list `n + n*(n-1)/4`. So, in the average case the complexity of insertion sort is also $O(n^2)$.

As a conclusion we can say that the time taken by insertion sort is essentially quadratic in the length of the list. In the exceptional case that the list is (almost) sorted, the time required is (almost) linear. The space required is always proportional to the length of the list, hence the space complexity is $O(n)$.

The quadratic time complexity of the insertion sort algorithm does not imply that sorting is always $O(n^2)$. We will now look at the time complexity of merge sort. In chapter 3 we implemented this sorting algorithm as:

```
msort :: [a] -> [a] | Ord a
msort xs
    | len<=1    = xs
    | otherwise = merge (msort ys) (msort zs)
where
    ys   = take half xs
    zs   = drop half xs
    half = len / 2
    len  = length xs
```

```
merge :: [a] [a] -> [a] | Ord a
merge []           ys   = ys
merge xs           []   = xs
merge p=:[x:xs] q=:[y:ys]
    | x <= y            = [x : merge xs q]
    | otherwise         = [y : merge p ys]
```

Computation of `len` takes `n` stapes, as usual `n` is the length of the list. The computation of `half` is done in a single step. The functions `take` and `drop` are both $O(n)$. Merging two lists takes $O(n)$ steps. So, one call of `merge` takes $O(n)$ steps plus two times the amount of steps required sorting a list of length `n/2`. This is $C(n) = 2 \times C(n/2) + c \times n$. This implies that merge sort is $O(n * \log n)$. The logarithm reflects the fact that a list of length `n` can be split $^2\log n$ times into lists of length greater than one. On average this algorithm has a better complexity than `isort`.

Our next sorting algorithm to study is quick sort. It is defined as:

```
qsort :: [a] -> [a] | Ord a
qsort []     = []
qsort [a:xs] = qsort [x \\ x<-xs | x<a] ++ [a] ++ qsort [x \\ x<-xs | x>=a]
```

In order to compute the complexity of this function we have to consider again various cases. Evaluating the list comprehensions takes $O(n)$ steps, where `n` is the length of the list to process. The first `++` takes a number of steps proportional to the length of the list generated by the first list comprehension. The second `++` takes only 2 steps. This is due to the fact that `++` associates to the right.

Now we consider the case that the input list is sorted. The first list comprehension will yield an empty list, which is sorted in a single reduction step. It takes one step to append the rest of the sorted list to this empty list. It requires $O(n)$ steps to reduce both list comprehensions. The length of the list to sort by the second recursive call of `qsort` is `n-1`. For the total amount of reductions steps we have. $C(n) = C(n-1)+2*n+3$. This implies that this 'best case' complexity of `qsort` is $O(n^2)$.

In the 'worst case' situation the list is sorted in the opposite order. The situation is very similar to the best case. The difference with the 'best case' is that all elements of the tail of the input end up in the first recursive call of `qsort` and the second list to sort is empty. Now the first `++` takes `n` reduction steps. This implies that $C(n) = C(n-1)+3*n+2$. The time complexity of the 'worst case' is also $O(n^2)$.

For the average case we assume that both list comprehensions yield a list containing half of the input list. The list comprehensions take both $O(n)$ reductions steps and the first `++` will consume `n/2` steps. This indicates that $C(n) = 2*C(n/2)+c*n$. Hence the `qsort` algorithm is in the average case $O(n * \log n)$. Note that the best case of the input for `isort` is not the best case for `qsort`. In contrast, best case for `isort`, is worst case for `qsort`.

The final sorting algorithm discussed in chapter 3 is tree sort.

```
tsort :: ([a] -> [a]) | Eq, Ord a
tsort = labels o listToTree

:: Tree a   = Node a (Tree a) (Tree a) | Leaf

listToTree :: [a] -> Tree a | Ord, Eq a
listToTree []     = Leaf
listToTree [x:xs] = insertTree x (listToTree xs)

insertTree :: a (Tree a) -> Tree a | Ord a
insertTree e Leaf   = Node e Leaf Leaf
insertTree e (Node x le ri)
    | e<=x  = Node x (insertTree e le) ri
    | e>x   = Node x le (insertTree e ri)
```

```
labels :: (Tree a) -> [a]
labels Leaf             = []
labels (Node x le ri)   = labels le ++ [x] ++ labels ri
```

One reduction step of `listToTree` is used for each element in the input list. For `insertTree` again three different cases are considered. When the list is random, the tree will becomes balanced. This implies that $^2\log n$ reduction steps are needed to insert an element in the tree. When the list is sorted, or sorted in reverse order, we obtain a degenerated (list-like) tree. It will take n reduction steps to insert the next element. The number of reduction steps for the function `labels` depends again on the shape of the tree. When all left sub-trees are empty, there are 3 reduction steps needed for every `Node`. This happens when the input list was inversely sorted. Since insertion is $O(n^2)$ in this situation, the complexity of the entire sorting algorithm is $O(n^2)$. When all right sub-trees are empty, the input was sorted, there are $O(n)$ reduction steps needed to append the next element. Since insertion is $O(n^2)$ in this situation, the entire algorithm is $O(n^2)$.

For balanced trees insertion of one element takes $O(^2\log n)$. Construction of the entire tree requires $O(n*\log n)$ steps. Transforming the tree to a list requires transforming two trees of half the size to lists, $2\times C(n-1)$, appending the second list to a list of n/2 elements. For the number of reduction steps we have $C(n) = 2\times C(n-1)+n/2+d$. Hence the to complexity of transforming the tree to a list is $O(n*\log n)$. This implies that tree sort has complexity $O(n*\log n)$.

Based on this analysis it is hard to say which sorting algorithm should be used. When you know that the list is almost sorted you can use `isort` and you should not use `qsort`. When you know that the list is almost sorted in the inverse order you can use `msort` and you should not use `isort` or `qsort`. For an list that is completely random `qsort` and `msort` a good choices. For an arbitrary list `msort` is a good choice. It is a little more expensive for a complete random list, but it behaves better for sorted lists.

Determining upper bounds and under bounds

Above we have determined the upper bounds of the complexity of various algorithms. When we do this carefully the obtained upper bound is also a tight upper bound. It is clear that this tight upper bound is not necessarily an under bound for the problem. Our first algorithm to reverse list is $O(n^2)$, while the second algorithm is $O(n)$. Also for the sorting algorithm we have a similar situation: the `isort` algorithm has complexity $O(n^2)$, while the exists clearly also sorting algorithms of $O(n*\log n)$.

The question arises whether it is possible to reverse a list in an number of reduction steps that is lower than $O(n)$. This is highly unlikely since we cannot imagine an algorithm that reverses a list without at least one reduction step for each element of the list. Reversing is $\Omega(n)$. Since we have an algorithm with complexity $O(n)$, the complexity of the best reversing algorithms will be $\Theta(n)$.

For sorting algorithms we can also determine an under bound. Also for sorting it is not feasible that there exists an sorting algorithm that processes each list element once. Sorting is at least $\Omega(n)$. We have not yet found a sorting algorithm with this complexity for an average list. Now have to decide whether we start designing better sorting algorithms, or to make a better approximation of the under bound of sorting. For sorting a general list it is not feasible that we can determining the desired position of an element by processing it once. The best we can hope for is that we can determine in which half of the list it should be placed. So, a better approximation of the under bound of sorting is $\Omega(n*\log n)$. Since we know at least one sorting algorithm with this complexity, we can conclude that sorting arbitrary lists is $\Theta(n*\log n)$.

Finding upper bounds of the complexity of an algorithm is not very difficult. When the approximations are made carefully, even determining close upper bounds of the algo-

rithm is merely a matter of counting. Finding tight upper bounds of the problem is more complicated, it involves a study of every algorithm feasible.

6.1.4 Constant factors

Above we have emphasised the complexity of problems and hence ignored all constant factors involved. This does not imply that constant factors are not important. The opposite is true, when efficiency is important you have to be keen on reduction steps that can be avoided, even if the overhead is just a constant factor. As a matter of fact, even a precise count of the reduction steps is not the final word, not every reduction steps in Clean takes an equal amount of time. So, some experiments can be very useful when the highest speed is wanted. See part III for additional information and hints.

The fact that `msort` is O(`n*log n`) and sorting is Θ(`n*log n`) does not imply that `msort` is the best sorting algorithm possible. It indicates that for large lists the increase of the time required is proportional to `n*log n`. For the actual execution time constant factors are important. When we look again at the function `msort`.

```
msort :: [a] -> [a] | Ord a
msort xs
    | len<=1    = xs
    | otherwise = merge (msort ys) (msort zs)
where
    ys   = take half xs
    zs   = drop half xs
    half = len / 2
    len  = length xs
```

It is clear that it can be improved with a constant factor. The functions `take` and `drop` both process the list `xs` until element `half`. This can be combined in a single function `split`:

```
msort2 :: ![a] -> [a] | Ord a
msort2 xs
    | len<=1    = xs
    | otherwise = merge (msort2 ys) (msort2 zs)
where
    (ys,zs) = split (len/2) xs
    len = length xs

split :: !Int [a] -> ([a],[a])
split 0 xs     = ([],xs)
split n [x:xs] = ([x:xs`],xs``)
where   (xs`,xs``) = split (n-1) xs
```

Further analysis shows that there is no real reason to compute the length of the list `xs`. This takes n steps. It is only necessary to split this list into two parts of equal length. This can also be done by selecting odd and even elements. Since we does not want to compute the length of the list to be sorted, the termination rules should also be changed. This is done in the function `msort3` and the accompanying function `split2`.

```
msort3 :: ![a] -> [a] | Ord a
msort3 []  = []
msort3 [x] = [x]
msort3 xs  = merge (msort3 ys) (msort3 zs)
where   (ys,zs) = split2 xs

split2 :: ![a] -> ([a],[a])
split2 [x,y:r] = ([x:xs],[y:ys]) where (xs,ys) = split2 r
split2 l       = (l,[])
```

Using accumulators we can avoid the construction of tuples to construct the parts of the list `xs`. In the function `msort4` we call the split function with empty accumulators.

```
msort4 :: ![a] -> [a] | Ord a
msort4 []  = []
msort4 [x] = [x]
msort4 xs  = merge (msort4 ys) (msort4 zs)
where   (ys,zs) = split3 xs [] []
```

```
split3 :: [a] [a] [a] -> ([a],[a])
split3 [x,y:r] xs ys = split3 r [x:xs] [y:ys]
split3 [x:r]   xs ys = (xs,[x:ys])
split3 l       xs ys = (xs,ys)
```

An other approach to avoid the computation of the length of the list to be sorted in each recursive call is to determine the length once and pass the actual length as argument to the actual sorting function. Since the supplied length and the actual length should be identical this approach is a little bit more error prone.

A similar technique can be used in the quick sort algorithm. Currently there are two list comprehensions used to split the input list. Using an additional function it is possible to do this in one pass of the input list. This is the topic of exercise 1.

Measurements: generating a pseudo random list

In order to investigate whether the reduced amount of reductions steps yields a more efficient algorithm we need to run the programs and measure the execution times. Since some of the sorting programs are sensitive to the order of elements in the input list we want to apply the sorting functions to a list of random numbers.

Due to the referential transparency random number generation is somewhat tricky in functional programs. When one random number is needed we can use for instance the seconds from the clock. However, when we fill a list of numbers in this way the numbers will be strongly correlated. The solution is to use *pseudo-random numbers.* Given a *seed* the next number can be generated by the *linear congruetial method.*

```
nextRan :: Int -> Int
nextRan s = (multiplier * s + increment) mod modulus
```

The constants `multiplier`, `increment` and `modulus` are suitable large numbers. In the examples below we will use the values:

```
multiplier  :== 26183
increment   :== 29303
modulus     :== 65536   //  this is 2^16
```

A sequence of these pseudo random numbers can be generated by:

```
ranList :: [Int]
ranList = iterate nextRan seed
```

The `seed` can be obtained for instance form the clock or be some constant. To compare the sorting functions we will use the constant `42`, this has the advantage that each sorting function has the same input.

The only problem with these numbers is the possibility of cycles. When `nextRan n = n`, or `nextRan (nextRan n) = n`, or `nextRan (nextRan ( …n)…) = n` for some `n`, there will be a cycle in the generated list of random numbers as soon as this `n` occurs once. When the constants are well chosen there are no troubles with cycles. In fact `nextRan` is an ordinary referential transparent function. It will always number the same value when the same seed is used. This implies that `nextList` will start a cycle as soon as an element occurs for the second time. It is often desirable that the same number can occur twice in the list of random numbers used without introducing a cycle. This can be achieved by scaling the random numbers.

When we needed random numbers in an other range we can scale the numbers in `ranList`. A simple appoach is:

```
scaledRans :: Int Int -> [Int]
scaledRans min max = [i mod (max-min+1) + min \\ i <- ranList]
```

When you need random numbers with a nice equal distribution over the whole range this function is only appopriate when `modulus` is a multiple of the range. Consider the situation where `modulus` is about 1.5 × range. Numbers in the first part of the range will occur twice as much as the other numbers. When `modulus` is much larger than the range, the ratio 2/1

changes to $n+1/n$. For large n this is a good approximation of 1. The problem can be avoided when by using a slightly more complex function to scale the random numbers.

```
scaledRans :: Int Int -> [Int]
scaledRans min max = [i \\ i <- [i/denominator + min \\ i <- ranList] | i<=max]
where denominator = modulus / (max-min+1)
```

The distrubution of these numbers is very well equal. When only the flatness of the distrubution count you can also use `flatten (repeat [min..max])`.

Measurements: sorting lists

In order to achieve a reasonable execution time[2] we sort a list of 1000 number 100 times and a list of 4000 elements 100 times. We prefer to sort a shorter list 100 times instead of a long list once in order to reduce the overhead of generating the input list and to reduce the amount of heap space necessary. By selecting one element of the list we prevent that a lot of time is spent on outputting the list. We select the last element in order to be sure that the entire list is sorted. This is achieved by the following program. At the position *sort* we substitute the name of the function to investigate. The `let!` construct is used to force evaluation.

```
Start = ntimes 100

ntimes n = let! e = last (sort rlist) in
           case n of
            0 -> e
            n -> ntimes (n-1)

rlist =: take 1000 ranList
```

The results are listed in the following table. As a reference we included also quick sort, tree sort and insertion sort. The functions `qsort2` and `tsort2` are variants of `qsort` and `tsort` introduced below. The result for the function `I` shows that the time to select the last element and to generate the list to process can be neglected. The function `msort2`` is the function `msort2` where `split` is replaced by the function `splitAt` from the standard environment.

function	1000 elements			4000 elements			ratio		
	execution	gc	total	execution	gc	total	execution	gc	total
`msort`	9.23	0.56	9.80	44.8	13.85	58.66	4.85	24.73	5.98
`msort2`	6.23	0.31	6.55	30.63	8.00	38.63	4.92	25.81	5.90
`msort2``	10.71	0.76	11.48	51.51	19.81	71.33	4.81	22.51	6.21
`msort3`	10.11	0.88	11.00	49.26	20.70	69.69	4.87	23.52	6.33
`msort4`	6.06	0.30	6.36	30.51	11.35	41.86	5.03	23.65	6.58
`qsort`	8.21	0.48	8.70	39.91	11.30	51.21	4.86	23.54	5.88
`qsort2`	6.51	0.30	6.81	31.81	6.83	38.65	4.89	22.77	5.67
`tsort`	9.28	0.53	9.81	38.11	9.70	47.81	4.11	18.30	4.87
`tsort2`	7.50	0.41	7.91	30.98	7.46	38.45	4.13	18.20	4.86
`isort`	105.08	3.85	108.93	1809.93	288.68	2098.61	17.22	74.98	19.27
`I`	0.11	0.00	0.11	0.46	0.00	0.46	4.18	-	4.18

Table 3: Execution-, garbage collection-, and total time in seconds of various sorting algorithms.

It is clear that any algorithm with complexity $O(n \log n)$ is much better on this size of input lists than `isort` with complexity $O(n^2)$. Although there is some difference between the various variants of the merge sort algorithm, it is hard to predict which one is the best. The difference between `msort2`` and `msort2` is caused by an optimisation for tuples that does not work across module boundaries.

[2] We used a Macintosh Performa 630 for the experiments. The application gets 1 MB of heap space and does use stack checks.

The complexity theory predicts that the ratio between the execution speed for the programs with complexity $O(n \log n)$ is 4.80, for an algorithm of $O(n^2)$ this ratio is 16. These numbers correspond pretty well with the measured ratios. Only the time needed by the tree sorts grows slower than expected. This only indicates that the list used are not large enough to neglect initialisation effects.

You can see also that the amount of garbage collection time required grows much faster than the execution time. The amount of garbage collection needed is determined by the number of nodes used during execution and the number of nodes that can be regained during garbage collection. For a large list, less memory can be regained during garbage collection, and hence more garbage collections are needed. This increases the total garbage collection time faster as you might expect based on the amount of nodes needed. To reduce the amount of garbage collection time the programs should be equipped with more memory.

For the user of the program only the total execution time matters. This takes both the pure reduction time as well as the time consumed by garbage collections into account. The total execution time is dependent of the amount of heap space used by the program.

An other thing that can be seen from this table is that it is possible to optimise programs by exploring the efficiency of some educated guesses. However, when you use a function with the right complexity it is only worthwhile to use a more complicated algorithm when the execution speed is a of prime interest. The difference is speed is between the various sorting algorithms is limited. We recommend to use one of the merge sort functions to sort list of an unknown shape. Quick sort and tree sort behave very well for random list, but for sorted list they are $O(n^2)$. This implies that the execution time will be much langer.

Other ways to speed up programs

An other way to speed up programs is by exploiting *sharing*. In the Fibonacci example above we saw that this can even change the complexity of algorithms. In the program used to measure the execution time of sorting functions we shared the generation of the list to be sorted. Reusing this lists of saves a constant factor for this program.

There are many more ways to speed up programs. We will very briefly mention two other possibilities. The first way to speed up a program is by executing all reduction steps that does not depend on the input of the program before the program is executed. This is called *partial evaluation* [Jones 95]. A way to achieve this effect is by using macro's whenever possible. More sophisticated techniques also look at applications of functions. A simple example illustrates the intention.

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

square :: Int -> Int
square x = power 2 x
```

Part of the reduction steps needed to evaluate an application of square does not depend on the value `x`. Using partial evaluation it is possible to transform the function `square` to

```
square :: Int -> Int
square x = x * x * 1
```

Using the mathematical law `x * 1 = x`, it is even possible to achieve:

```
square :: Int -> Int
square x = x * x
```

The key idea of partial evaluation is to execute rewrite steps that does not depend on the arguments of a function before the program is actually executed. The partially evaluated program will be faster since the number of reduction steps needed to execute is lower.

The next technique to increase the efficiency of programs to combine the effects of several functions to one function. This is called *function fusion*. We will illustrate this with the function `qsort` as example. This function was defined as:

```
qsort :: [a] -> [a] | Ord a
qsort []     = []
qsort [a:xs] = qsort [x \\ x<-xs | x<a] ++ [a] ++ qsort [x \\ x<-xs | x>=a]
```

At first glance there is only one function involved here. A closer look shows that also the operator `++` is to be considered as a function. As a matter of fact also the list comprehensions can be seen as functions. The compiler will transform them to ordinary functions. We will restrict ourselves here to the function `qsort` and the operator `++`. When `qsort` has sorted some list of elements smaller than the first element, the operator `++` scans that list in order to append the other elements of the list. During the scanning of the list all cons nodes will be copied. This is clearly a waste of work.

This can be avoided by using a *continuation* argument in the sorting function. The continuation determines what must be done when this function is finished. In this example what must be appended to the list when the list to be sorted is exhausted. Initially there is nothing to be appended to the sorted list. We use an additional function, `qs`, that handles sorting with continuations.

```
qsort2 :: [a] -> [a] | Ord a
qsort2 l = qs l []
```

The continuation of sorting all elements smaller than the first element is the sorted list containing all other elements. Its continuation is the continuation supplied as argument to `qs`. When the list to be sorted is empty we continue with the continuation `c`.

```
qs :: [a] [a] ->[a] | Ord a
qs []     c = c
qs [a:xs] c = qs [x \\ x<-xs | x<a] [a:qs [x \\ x<-xs | x>=a] c]
```

The trace of the reduction of a small example clarifies the behaviour of these functions:

```
qsort2 [1,2,1]
→   qs [1,2,1] []
→   qs [] [1:qs [2,1] []]
→   [1:qs [2,1] []]
→   [1:qs [1] [2:qs [] []]]
→   [1:qs [] [1:qs [] [2:qs [] []]]]
→   [1:[1:qs [] [2:qs [] []]]]
→   [1:[1:[2:qs [] []]]]
→   [1:[1:[2:[]]]]
=   [1,1,2]
```

```
qsort [1,2,1]
→   qsort []++[1]++qsort [2,1]
→   []++[1]++qsort [2,1]
→   [1]++qsort [2,1]
→   [1:[]++qsort [2,1]]
→   [1:qsort [2,1]]
→   [1:qsort [1]++[2]++qsort []]
→   [1:qsort []++[1]++qsort []++
                          [2]++qsort []]
→   [1:[]++[1]++qsort []++[2]++qsort []]
→   [1:[1]++qsort []++[2]++qsort []]
→   [1:1:[]++qsort []++[2]++qsort []]
→   [1:1:qsort []++[2]++qsort []]
→   [1:1:[]++[2]++qsort []]
→   [1:1:[2]++qsort []]
→   [1:1:2:[]++qsort []]
→   [1:1:2:qsort []]
→   [1:1:2:[]]
=   [1,1,2]
```

Traces showing the advantage of using continuations in `qsort2` over the ordinary function `qsort`.

It is obvious that the version of quick sort using continuations requires a smaller number of reduction steps. This explains why it is 25% faster.

In the same spirit we can replace the operator `++` in the tree sort function, `tsort` by a continuation.

```
tsort2 :: ([a] -> [a]) | Eq, Ord a
tsort2 = labels2 [] o listToTree

labels2 :: [a] (Tree a) -> [a]
labels2 c Leaf           = c
labels2 c (Node x le ri) = labels2 [x: labels2 c ri] le
```

As shown in the table of execution times above, this function using continuations is indeed (about 20%) more efficient.

6.1.5 Exploiting Strictness

As explained in previous chapters, an function argument is strict when its value is needed always in the evaluation of a function call. Usually an expression is not evaluated until its value is needed. This implies that expressions causing nonterminating reductions or errors and expressions yielding infinite data structures can be used as function argument. Problems do not arise until the value of such expressions is really needed.

The price we have to pay for lazy evaluation is a little overhead. The graph representing an expression is constructed during a rewrite step. When the value of this expression is needed the nodes in the graph are inspected and later on the root is updated by the result of the rewrite process. This update is necessary since the node many be shared (occurring at several places in the expression). By updating the node in the graph recomputation of its value is prevented.

When the value of a node is know to be needed it is slightly more efficient to compute its value right away and store the result directly. The subexpressions that are known to be needed anyway are called *strict*. For these expressions there is no reason to store the expression and to delay its computation until it is needed. The Clean compiler uses this to evaluate strict expressions at the moment they are constructed. This does not change the number of reduction steps. It only makes the reduction steps faster.

The Clean compiler uses basically the following rules to decide whether an expression is strict:

1) The root of the right-hand side is a strict expression. When a function is evaluated this is done since its value is needed. This implies that also the value of its reduct will be needed. This is repeated until the root of the right hand side cannot be reduced anymore.
2) Strict arguments of functions occurring in a strict context are strict expressions. The function is known to be needed since it occurs in a strict context. In addition it is known that the value of the strict arguments is needed when the result of the function is needed.

These rules are recursively applied to determine as much strict subexpressions as possible. This implies that the Clean compiler can generate more efficient programs when strictness of function arguments is known. Generally strictness is an undecidable property. We does not make all arguments strict in order to be able to exploit the advantages of lazy evaluation. Fortunately, any safe approximation of strictness helps to speed up programs. The Clean compiler itself is able to approximate the strictness of function arguments. The compiler uses a sophisticated algorithm base on abstract interpretation [Plasmeijer 94]. A simpler algorithm uses the following rules:

1) Any function is strict in the first pattern of the first alternative. The corresponding expression should be evaluated in order to determine whether this alternative is applicable. This explains why the append operator, `++`, is strict in its first argument.

   ```
   (++) infixr 5 :: ![x] [x] -> [x]
   (++) [hd:tl] list = [hd:tl ++ list]
   (++) nil     list = list
   ```

   Since it is generally not known how much of the generated list is needed, the append operator is not strict in its second argument.
2) A function is strict in the arguments that are needed in all of its alternatives. This explains why the function `add` is strict in both of its arguments and `mul` is only strict in its first argument. In the standard environment both `+` and `*` are defined to be strict in both arguments.

```
mul :: !Int Int -> Int;
mul 0 y = 0
mul x y = x*y

add :: !Int !Int -> Int;
add 0 y = y
add x y = x+y
```

You can increase the amount of strictness in your programs by adding strictness information to function arguments in the type definition of functions. Subexpressions that are known to be strict, but does not correspond to function arguments can be evaluated strict by defining them as strict local definitions using `#!` or `let!`.

6.1.6 Unboxed values

Objects that are manipulated inside the program as plain value instead of being embedded inside a node in the heap are called *unboxed* values. These unboxed values are handled very efficiently by the Clean system. In this situation the Clean system is able to avoid the general graph transformations prescribed in the semantics. It is the responsibility of the compiler to use unboxed values and to do the conversion with nodes in the heap whenever appropriate. Strict arguments of a basic type are handled as unboxed value in Clean. Although the compiler takes care of this, we can use this to speed up our programs by using strict arguments of a basic type whenever appropriate.

We illustrate the effects using the familiar function length. A naive definition of `Length` is:

```
Length :: ![x] -> Int
Length [a:x] = 1 + Length x
Length []    = 0
```

A trace shows the behaviour of this function:

```
Length [7,8,9]
→   1 + Length [8,9]
→   1 + 1 + Length [9]
→   1 + 1 + 1 + Length []
→   1 + 1 + 1 + 0
→   1 + 1 + 1
→   1 + 2
→   3
```

The Clean system builds an intermediate expression of the form `1 + 1 + …+ 0` of a size proportional to the length of the list. Since the addition is know to be strict in both arguments, the expression is constructed on the stacks rather then in the heap. Nevertheless it takes time and space.

Construction of the intermediate expression can be avoided using an accumulator: a counter indicating the length of the list processed until now.

```
LengthA :: ![x] -> Int
LengthA l = L 0 l
where   L :: Int [x] -> Int
        L n [a:x] = L (n+1) x
        L n []    = n
```

The expression `Length [7,8,9]` is reduced as:

```
LengthA [7,8,9]
→   L 0 [7,8,9]
→   L (0+1) [8,9]
→   L ((0+1)+1) [9]
→   L (((0+1)+1)+1) []
→   ((0+1)+1)+1
→   (1+1)+1
→   2+1
→   3
```

The problem with this definition is that the expression used as accumulator grows during the processing of the list. Evaluation of the accumulator is delayed until the entire list is processed. This can be avoided by making the accumulator strict.

```
LengthSA :: ![x] -> Int
LengthSA l = L 0 l
where   L :: !Int [x] -> Int
        L n [a:x] = L (n+1) x
        L n []    = n
```

In fact the Clean system itself is able to detect that this accumulator is strict. When you don't switch strictness analysis off the Clean system will transform `LengthA` to `LengthSA`. The trace becomes:

```
LengthSA [7,8,9]
→   L 0 [7,8,9]
→   L (0+1) [8,9]
→   L 1 [8,9]
→   L (1+1) [9]
→   L 2 [9]
→   L (2+1) []
→   L 3 []
→   3
```

Since the accumulator is a strict argument of a basic type, the Clean system avoids the construction of datastructures in the heap. An unboxed integer will be used instead of the nodes in the heap. In table 4 we list the run time of some programs to illustrate the effect of strictness. We used a Power Macintosh 7600 to compute 1000 times the length of a list of 10000 elements. The application had 400 KB of heap. The difference between the programs is the function used to determine the length of the list.

function	execution	gc	total
`Length`	5.65	0.01	5.66
`LengthA`	20.76	86.45	107.21
`LengthSA`	2.70	0.0	2.70

Table 4: Runtime in seconds of a program to determine the length of a list.

Adding a lazy accumulator has a severe runtime penalty. This is caused by that fact that all computations are now done in a lazy context. The intermediate expresion `1+1+…+0 is` constructed in the heap. Adding the appropriate strictness information makes the function to compute the length of a list twice as efficient as the naive definition. Adding this strictness information improves the efficiency of the computation of the length of a list using an accumulator by a factor of 40. The overloaded version of this function defined in StdEnv does use the efficient algorithm with a strict accumulator.

Adding a strictness annotation can increase the efficiency of the manipulation of basic types significantly. You might even consider adding strictness annotations to arguments that are not strict in order to increase the efficiency. This is only allowed when you know that the corresponding expression will terminate.

As example we consider the function to replace a list of items by a list of their indices:

```
indices :: [x] -> [Int]
indices l = i 0 l
where   i :: Int [x] -> [Int]
        i n []    = []
        i n [a:x] = [n: i (n+1) x]
```

The local function `i` is not strict in its first argument. When the list of items is empty the argument `n` is not used. Nevertheless, the efficiency of the function `indices` can be doubled (for a list of length 1000) when this argument is made strict by adding an annotation. The cost of this single superfluous addition is outweighed by the more efficient way to handle this argument.

We have seen an other example in the function `fib4` to compute Fibonacci numbers in linear time:

```
fib4 n = f n 1 1
where   f :: !Int !Int !Int -> Int
        f 0 a b = a
        f n a b = f (n-1) b (a+b)
```

Making `f` strict in its last argument does cause that one addition is done too much (in the last iteration of `f` the last argument will not be used), but makes the computation of `fib4 45` twelve times as efficient. When `f` evaluates all its arguments lazy, the Fibonacci functions slows down by an other factor of two.

6.1.7 The cost of Currying

All functions and constructors can be used as Curried function is Clean. Although you are encouraged to do this whenever appropriate, there are some runtime costs associated with Currying. When speed becomes an issue it may be worthwhile to consider the elimination of some heavily used Curried functions from you program.

The cost of Currying are caused by the fact that it is not possible to detect at compile time which function is applied and whether it has the right number of arguments. This implies that this should be done at runtime. Moreover certain optimisations cannot be applied for Curried functions. For instance, it is not possible to use unboxed values for strict arguments of basic types. The Clean system does not know which function will be applied. Hence, it cannot be determined which arguments will be strict. This causes some additional loss of efficiency compared with a simple application of the function.

To illustrate this effect we consider the function `sum` to compute the sum of a list of integers. The naive definition is:

```
Sum :: ![Int] -> Int
Sum [a:x] = a + Sum x
Sum []    = 0
```

Using the appropriate fold function this can be written as `Foldr` (+) 0. Where `Foldr` is defined as:

```
Foldr :: (a b -> b) b ![a] -> b
Foldr op r [a:x] = op a (Foldr op r x)
Foldr op r []    = r
```

In the function `Sum` the addition is treated as an ordinary function. It is strict in both arguments and the arguments are of the basic type `Int`. In the function `Foldr` the addition is a curried function. This implies that the strictness information cannot be used and the execution will be slower. Moreover it must be checked whether op is a function, or an expression like `I (+)` that yields a function. Also the number of arguments needed by the function should be checked. Instead of the ordinary addition there can be a weird function like `\n -> (+) n`, this function takes one of the arguments and yield a function that takes the second argument. Even when these things does not occur, the implementation must cope with the possibility that it is there at runtime. For an ordinary function application, it can be detected at compile time whether there is an ordinary function application.

The function `foldr` from the standard environment eliminates these drawbacks by using a macro:

```
foldr op r l :== foldr r l
    where   foldr r []      = r
            foldr r [a:x]   = op a (foldr r x)
```

By using this macro a tailor made `foldr` is created for each and every application of `foldr` in the text of your Clean program. In this tailor made version the operator can usually be treated as an ordinary function. This implies that the ordinary optimisations will be applied.

As argued above, it is better to equip the function to sum the elements of a list with an accumulator.

```
SumA :: ![Int] -> Int
SumA l = S 0 l
where   S :: !Int ![Int] -> Int
        S n [a:x] = S (n+a) x
        S n []    = n
```

The accumulator argument `n` of the function `SumA` is usually not considered to be strict. Its value will never be used when `SumA` is applied to an infinite list. However, the function `SumA` will never yield a result in this situation.

The same recursion pattern is obtained by the expression `Foldl (+) 0`. This fold function can be defined as:

```
Foldl :: (b a -> b) !b ![a] -> b
Foldl op r [a:x] = Foldl op (op r a) x
Foldl op r []    = r
```

The second argument of this function is made strict exactly for the same reason as in `SumA`. In StdEnv also this function is defined using a macro to avoid the cost of Currying:

```
foldl op r l :== foldl r l
    where   foldl r []      = r
            foldl r [a:x]   = foldl (op r a) x
```

We will compare the run time of programs computing 1000 times the sum of the list `[1..1000]` in order to see the effects on the efficiency.

function	execution	gc	total
`Sum`	6.00	0.00	6.00
`Foldr (+) 0`	30.13	7.53	37.66
`foldr (+) 0`	6.01	0.00	6.01
`SumA`	2.51	0.00	2.51
`Foldl (+) 0`	19.66	2.55	22.21
`foldl (+) 0`	2.68	0.01	2.68

Table 5 Runtime in seconds of a program to determine the costs of Currying.

The table shows the impact of omitting all strictness information, also the strictness analyser of the Clean system is switched off. The only remaining strictness information is the strictness of the operator `+` from StdEnv.

function	execution	gc	total
`Sum`	6.03	0.00	6.03
`Foldr (+) 0`	29.13	7.75	37.41
`foldr (+) 0`	6.13	0.00	6.13
`SumA`	16.13	3.70	19.83
`Foldl (+) 0`	37.86	7.16	45.03
`foldl (+) 0`	15.61	3.50	19.11

Table 6 Runtime in seconds of a program to determine the costs of Currying without strictness.

From these tables we can conclude that there are indeed quite substantial costs involved by using Curried functions. However, we used a Curried function manipulating strict arguments of a basic type here. The main efficiency effect is caused by the loose of the possibilities to treat the arguments as unboxed values. For functions manipulating ordinary datatypes the cost of Currying are much smaller. When we use the predefined folds from StdEnv there is no significant overhead in using Curried functions due to the macro's in the definition of these functions.

6.1.8 A word of warning

In this section we have shown that it is possible to reason about the number of reduction steps needed to execute a program. In addition we have introduced some techniques to reduce the amount of reduction steps and the cost of reduction steps. This does not mean that you have to squeeze the last reduction step out of you program and to avoid laziness or curried functions. We just want to show that there are some cost associated with these language constructs and what can be done to reduce these cost when the (lack of) execution speed is a problem.

Your computer is able to do a lot of reduction steps (up to several million) each second. So, usually it is not worthwhile to eliminate all possible reduction steps. Your program should in the first place be correct and solve the given problem. The readability and maintainability of your program is often much more important than the execution speed. Programs that are clear are more likely to be correct and better suited for changes. Too much optimisation can be a real burden when you have to understand or change programs. The complexity of the algorithms in your program can be a point of concern.

In our treatment of complexity we have ignored lazy evaluation. Lazy evaluation complicates accurate determination of the number of reductions steps severely. We have always assumed that the entire expression should be evaluated. In the examples we have taken care that this is what happens. However, when we select the first element of a sorted list it is clear that the list will not be sorted entirely due to lazy evaluation. Nevertheless a lot of comparisons are needed that prepares for sorting the entire list. The given determination of the complexity remains valid as an upper bound. Determining the under bound, or accurate number of reduction steps, is complicated by lazy evaluation.

6.2 A guide to local definitions and scopes

In the previous chapters we have seen several ways to define local functions. Since local functions without argument are handled differently the are usually called graphs. Each of these methods is introduced with a special purpose in mined. Here we will list the possibilities for making local definitions. We will indicate why this possibility exists and show its scope.

Local functions are used to limit the scope of the function. Moreover the arguments of the surrounding function and the local defined graphs can be used without the obligation to pass them explicitly as argument to the local function. Graphs are used to share expressions and to give an expression a name to enhance the readability of the program.

6.2.1 Local definitions

With a *where* block one can define functions and graphs which have meaning within every expression appearing in a *function alternative.* This is the default way to introduce local functions and graphs. It is not possible to define functions or graphs local to a whole function definition (i.e. in scope of all function alternatives of a function definition).

```
function args
         | guard1=expression1
         | guard2 = expression2
         where
             selector = expression
             function args = body
```

Figure 1: Defining functions and graphs locally for a function alternative.

The scope induced by a *where* block can sometimes be too big. In Clean one can also restrict the scope of definitions to the body of a *rule alternative* by using a *with* statement.

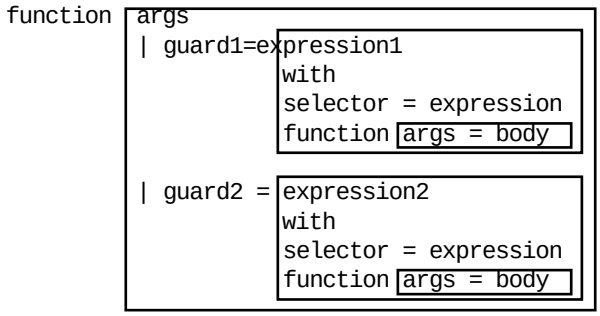

Figure 2: Defining functions and graphs locally for a rule alternative.

With a *let* statement one can define new functions and graphs which only have a meaning within a certain expression. This is allowed in any expression on the right-hand side of a function or graph definition. Let blocks are allowed also in sub expressions.

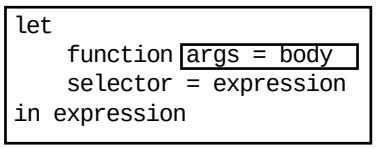

Figure 3: Defining functions and graphs locally for a certain expression.

A closely related form is the *let!* statement. The same scope rule applies for a *strict let* expression. The difference with the ordinary let expressions is that only a single identifier may be defined and that the evaluation of this identifier is forced. The identifier behaves as the strict arguments of a function. This expression is only allowed at the root of the right-hand side. It is for instance useful to select an element from an unique data structure before it is updated.

```
DoubleArrayElem :: *{Int} Int -> .{Int}
DoubleArrayElem a i = let! e = a.[i]
                      in {a & [i] = e + e}
```

For expressions which have to be evaluated in a certain sequential order it is very convenient to define these expressions *before* a guard in which they can be tested. This makes it possible to define selectors (graphs) and tests on the contents of these selectors in a textual order which closely corresponds to the order in which they are supposed to be evaluated. A special *let* statement is provided (keyword `let` or `#`) called *let-before* which introduces a very special scope. Each selector defined induces a new scope (excluding the body of the selector, see the picture below) ending at the function body.

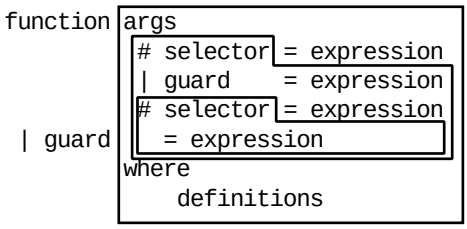

Figure 4: Defining local graphs before a guard.

It is possible to use the same name in several let-before definitions. This is very convenient for environment passing or objects that should be passed around ion a single treated way. The object keeps its name. More actually, each version of the object has its own name. All these names happen to be equal. This is illustrated in the function `readchars` that reads all characters from an unique file.

```
readchars :: *File -> ([Char], *File)
readchars file
```

```
#  (ok,char,file)  = freadc file
|  not ok          = ([],file)
#  (chars,file)    = readchars file
=  ([char:chars], file)
```

Functions and graphs defined in the where block can be used in let-before expressions. One cannot use the identifiers defined in a let-before in a where block. Allowing the use of identifiers defined in the let-before block in the where block can introduce cyclic dependencies. This is problematic since it is possible, and often useful, to define several identifiers with the same name in the let-before block. Using definitions in the where block makes it either possible to spoil the nested semantics of the let-before block or cause very strange semantic rules. To avoid these problems identifiers from the let-before block cannot be used in the where block. When you need a local definition containing identifiers of the let-before block you should use a with block.

There exists also a strict version of the let-before, `#!`. This isn't heavily used since the single treatedness is often more important than the exact moment of execution. Also the use of the value of the identifiers defined in a let before forces the evaluation. In the example above, the guard `not ok` forces the evaluation of `freadc file`.

6.2.2 Scope within expressions

Within an expression new formal parameters can also be introduced. This can happen in a *lambda expression*, which is a nameless function. The formal parameters have a meaning in the corresponding function body.

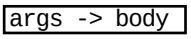

Figure 5: Scope of lambda expression.

Local definitions in a lambda expression can be introduced through a let expression. A ridiculuous example is e very complex identity function:

```
difficultIdentity :: !a -> a
difficultIdentity x = (\y -> let z = y in z) x
```

New formal parameters can also be introduced in a *case expression*. They have a meaning in the corresponding case alternative identical to the scope rules of an ordinary function definition.

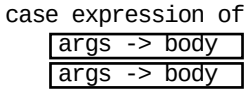

Figure 6: Scopes in a case expression.

This implies that the `args` from a case cannot be used in a where block. When you want to introduce functions or graphs in one of the bodies of a case expression you should introduce them using `let` or `with`.

As example we show the function `sumElementsToCome` that transforms a list of elements to a list of tuples containing the sum of the elements to come and the element. The value of the expression `sumElementsToCome [1..3]` is `[(5,1),(3,2),(0,3)]`.

```
sumElementsToCome :: [x] -> [(x,x)] | +,zero x
sumElementsToCome l
 = case l of [a:x] | isEmpty x -> [(zero,a)]
                   | otherwise -> [(b+c,a):y]
                     with y = sumElementsToCome x; [(b,c):_] = y
             []  -> []
```

In a *list* and in an *array comprehension* new variables can be introduced when *generators* are specified. Each generator can generate a selector which can be tested in a guard and used to generate the next selector and finally in the resulting expression (see figure 7).

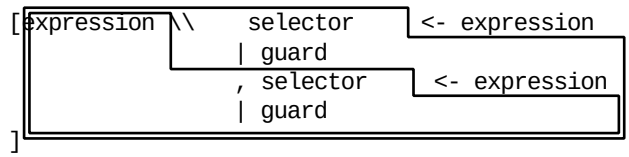

Figure 7: Scopes in a list comprehension.

This enables us to write the function to flatten a list (append all its elements) as:

```
Flatten :: ![[a]] -> [a]
Flatten lists = [elem \\ list <- lists, elem <- list]
```

All Pythagorean triangle with sides less or equal than 100 are generated by:

```
triangles :: [(Int,Int,Int)]
triangles = [ (a,b,c)   \\ a <- [1..max]
                        ,  b <- [a..max]
                        ,  c <- [b..max]
                        |  a^2 + b^2 == c^2
                        ]
            where max = 100
```

By using the list generators `[a..max]` and `[b..max]` we prevented that permutations of triangles are found.

6.3 Equational reasoning

Using proofs you can argue about the behaviour of a program over all its inputs. Testing can only observe the behaviour of a finite set of inputs. Although you can choose the test set to be representative for the total behaviour, testing is generally more limited. Here we will reason about the correctness of functions.

Direct proofs

The simplest form of proofs are direct proofs. A direct proof is a obtained by a sequence of rewrite steps. For a simple example we consider the following definitions:

```
I :: t -> t
I x = x             // This function is defined in StdEnv.

twice :: (t->t) t -> t
twice f x = f (f x) // This function is defined in StdEnv.

f :: t -> t
f x = twice I x
```

When we want to show that `f x = x` for all `x`, we can run a lot of tests. However, there are infinitely many possible arguments for `f` . So, testing can build confidence, but can't show that truth of `f x = x`. A simple proof shows that `f x = x` for all `x`. We start with the function definition of `f` and apply reduction steps to its body.

```
f x = twice I x      // The function definition
    = I (I x)        // Using the definition of twice
    = I x            // Using the definition of I for the outermost function I
    = x              // Using the definition of I                              □
```

This example shows the style we will use for proofs. The proof consists of a sequence of equalities. We will give a justification of the equality as a comment and end the proof with the symbol □.

Even direct proofs are not always as simple as the example above. The actual proof consists usually of a sequence of equalities. The crux of constructing proofs is to decide which equalities should be used.

For the same functions it is possible to show that the functions `f` and `I` behave equal. It is tempting to try to prove `f = I`. However, we won't succeed when we try to proof the function `f` equal to `I` using the same technique as above. It is not necessary that the function

bodies can be shown equivalent. It is sufficient that we show that functions `f` and `I` produce the same result for each argument: `f x = I x`. In general: *two functions are considered to be equivalent when they produce the same answer for all possible arguments.* It is very simple to show this equality for our example:

```
f x = twice I x      // The function definition
    = I (I x)        // Using the definition of twice
    = I x            // Using the definition of I for the outermost function I
```
□

As you can see from this example it is not always necessary to reduce expressions as far as you can (to normal form). In other proofs it is needed to apply functions in the opposite direction: e.g. to replace `x` by `I x`.

A similar problem arises when we define the function `g` as:

```
g :: (t -> t)
g = twice I
```

And try to prove that `g x = x` for all `x`. We can't start with the function definition and apply rewrite rules. In order to show this property we have to supply an arbitrary argument `x` to the function `g`. After invention of this idea the proof is simple and equivalent to the proof of `f x = x`.

Proof by cases

When functions to be used in proof are defined consists of various alternatives or contain guards its is not always possible to use a single direct proof. Instead of one direct proof we use a direct proof for all relevant cases.

As an example we will show that `abs x ≥ 0` for all `x`, using

```
abs :: Int -> Int
abs n | n < 0     = ~n
      | otherwise = n
```

Without assumptions on the argument the proof can't be made. The cases to distinguish are indicated by the function under consideration. Here the guard determines the cases to distinguish.

Proof

Case `x < 0`

```
abs x = ~x           // Using the definition of abs
```

Since `x < 0`, `~x` will be greater than `0`, using the definitions in the class `Eq`, `x > 0` implies `x ≥ 0`. Hence, `abs x ≥ 0` when `x < 0`.

Case `x ≥ 0`

```
abs x = x            // Using the definition of abs
```

Since `x ≥ 0` and `abs x = x` we have `abs x ≥ 0`.

Each `x` is either `< 0` or `≥ 0`. For both cases we have `abs x ≥ 0`. So, `abs x ≥ 0` for all `x`. □

The last line is an essential step in the proof. When we do not argue that we have covered all cases there is in fact no proof. Nevertheless, this last step is often omitted. It is fairly standard and it is supposed to be evident for the reader that all cases are covered.

The trouble of proving by cases is that you have to be very careful to cover all possible cases. A common mistake for numbers is to cover only the cases `x<0` and `x>0`. The case `x=0` is erroneously omitted.

Proof by case can be done for any data type. Sometimes we can handle many values at once (as is done in the proof above), in other situation we must treat some or all possible values separately. Although a proof by case can have many cases, the number of cases should at least be finite.

As additional example we show that `Not (Not b) = b` for all Booleans b using:

```
Not :: Bool -> Bool
Not True  = False
Not False = True
```

Proof of `Not (Not b) = b`.

Case `b == True`

```
Not (Not b)             // The value to be computed.
    = Not (Not True)    // Using the assumption of this case.
    = Not False         // Using the definition of Not for the innermost application.
    = True              // Using the definition of Not.
```

Case `b == False`

```
Not (Not b)             // The value to be computed.
    = Not (Not False)   // Using the assumption of this case.
    = Not True          // Using the definition of Not for the innermost application.
    = False             // Using the definition of Not.
```

Each Boolean (`Bool`) is either `True` or `False`. So, this proof covers all cases and proves that `Not (Not b) = b` for all Booleans `b`. □

Proof by induction

As stated in the previous section proof by cases works only when there are a finite amount of cases to be considered. When there are in principle infinitely many cases to consider we can often use a proof by induction. The principle of proving properties by induction is very well known in mathematics. In mathematics we prove that some property `P(n)` holds for all natural numbers `n` by showing two cases:

Base case — Prove `P(0)`

Induction step — Prove `P(n+1)` assuming that `P(n)` holds.

The principle of this proof is very similar to recursion. Using the base case and the induction step we can prove `P(1)`. Using `P(1)` and the induction step we show that `P(2)` holds. In the same way we can prove `P(3)`, `P(4)`, `P(5)`, ... Using this machinery we can prove `P(n)` for any `n`. Since this is a common proof method it suffices to show the base case and the induction step. The fact that the property can be proven for any value from these parts is taken for granted when you refer to induction.

As example we will show that the following efficient definition of the Fibonacci function and the naive definition are equivalent for all non-negative integers.

```
tupleFib :: Int -> Int
tupleFib n = fibn
    where   (fibn,_) = tf n
            tf 0 = (1,1)
            tf n = (y,x+y) where (x,y) = tf (n-1)

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

Proof of `tupleFib n = fib n`

The key step here is to understand that `tf n = (fib n, fib (n+1))` for all `n`. Once we have seen that this is the goal of our proof, it can be proven by induction.

Case `n == 0`

We have to prove that `tf 0 == (fib 0, fib (0+1))`. This is done by rewriting the expression `tf 0`.

```
tf 0                    // Expression to be proven equal to (fib 0, fib (0+1))
    = (1,1)             // Definition of tf.
    = (fib 0,1)         // Using first alternative of the definition of fib.
    = (fib 0,fib 1)     // Using second alternative of the definition of fib.
    = (fib 0,fib (0+1)) // Arithmetic.
```

This proves that `tf n = (fib n, fib (n+1))` for `n = 0`.

Case `n + 1`

We have to show that `tf (n+1) = (fib (n+1), fib (n+1+1))` assuming that `tf n = (fib n, fib (n+1))`. We will prove this by rewriting the expression `tf (n+1)`.

```
tf (n+1)                                          // Initial expression.
    = (y,x+y) where (x,y) = tf (n+1-1)            // Definition of tf, assuming that n>0.
    = (y,x+y) where (x,y) = tf n                  // Arithmetic
    = (y,x+y) where (x,y) = (fib n, fib (n+1))    // Induction hypothesis
    = (fib (n+1),fib n + fib (n+1))               // Rewriting the expression.
    = (fib (n+1),fib (n+1+1))                     // Last alternative of function fib.
```

This proves that `tf (n+1) = (fib (n+1), fib (n+1+1))` assuming that `tf n = (fib n, fib (n+1))`. These case together prove by induction that `tf n = (fib n, fib (n+1))` for all positive `n`. Using this result, proving `tupleFib n = fib n` is done by a direct proof.

```
tupleFib n
    = fibn where (fibn,_) = tf n                  // Definition of tupleFib.
    = fibn where (fibn,_) = (fib n, fib (n+1))    // Using the result proven above.
    = fib n                                       // Rewriting the expression.      □
```

The key step in designing proofs is to find the appropriate sub-goals. Proofs can become very complex, by having many sub-goals. These sub-goals can require additional induction proofs and sub-goals, and so on.

As additional example we will prove the following Fibonacci function that avoids tuples equivalent to the function `fib`.

```
fastFib :: Int -> Int
fastFib n = f n 1 1
            where f 0 a b = a
                  f n a b = f (n-1) b (a+b)
```

Proof of `fastFib n = fib n` for all `n ≥ 0`.

We will first prove that `f n 1 1 = f (n-m) (fib m) (fib (m+1))` for all `m` such that `m≥0 & m≤n`. This is proven again by induction:

Proof of `f n 1 1 = f (n-m) (fib m) (fib (m+1))`

Case `m == 0` of `f n 1 1 = f (n-m) (fib m) (fib (m+1))`

```
f (n-m) (fib m) (fib (m+1))                       // Start with right-hand side.
    = f n (fib 0) (fib 1)                         // Use m == 0.
    = f n 1 1                                     // Use function fib.
```

Case `m + 1` of `f n 1 1 = f (n-m) (fib m) (fib (m+1))`

```
f n 1 1                                           // Left-hand side of goal.
    = f (n-m) (fib m) (fib (m+1))                 // Use induction hypothesis.
    = f (n-(m+1)) (fib (m+1)) (fib m + fib (m+1)) // Rewrite according to f.
    = f (n-(m+1)) (fib (m+1)) (fib ((m+1)+1))     // Using fib in reverse.
```

This proves that `f n 1 1 = f (n-m) (fib m) (fib (m+1))`. Now we can use this result to prove that `fastFib n = fib n`.

```
fastFib n                                         // Left-hand side of equality to prove.
    = f n 1 1                                     // Definition of fastFib.
    = f (n-m) (fib m) (fib (m+1))                 // Using the sub-goal proven above.
    = f 0 (fib n) (fib (n+1))                     // Use m = n and arithmetic.
    = fib n                                       // According to definition of f.  □
```

There is no reason to limit induction proofs to natural numbers. In fact induction proofs can be given for any ordered (data)type. A well-known example in functional languages is the data type list. The base case is `P([])`, the induction step is `P([x:xs])` assuming `P(xs)`.

As example we will show the equivalence of the following functions to reverse lists. The function `rev` is simple and a clear definition. In chapter III.3 we will show that `reverse` is more efficient.

```
rev :: [t] -> [t]                   // Quadratic in the length of the argument list.
rev []    = []
rev [a:x] = rev x ++ [a]

reverse :: [t] -> [t]               // Linear in the length of the argument list.
reverse l = r l []
    where   r []    y = y
            r [a:x] y = r x [a:y]
```

Proof of `rev l = reverse l` for every list `l`.

In order to prove this we first prove an auxiliary equality: `r xs ys = reverse xs ++ ys`. This is proven by induction to `xs`.

Proof of `r xs ys = rev xs ++ ys`.

Case `xs == []` of `r xs ys = rev xs ++ ys`.

```
r xs ys
    = r [] ys                       // Using xs == [].
    = ys                            // Definition of the function r.
    = [] ++ ys                      // Definition of the operator ++ in reverse.
    = rev [] ++ ys                  // The alternative of rev in reverse.
```

Case `xs == [a:x]` of `r xs ys = rev xs ++ ys`. Assuming `r x ys = rev x ++ ys`.

```
r xs ys
    = r [a:x] ys                    // Using xs == [a:x].
    = r x [a:ys]                    // According to function r.
    = rev x ++ [a:ys]               // Using the induction hypothesis
    = rev x ++ [a] ++ ys            // Using operator ++ in reverse.
    = rev [a:x] ++ ys               // Using associativity of ++ and last alternative of rev.
```

This proves `r xs ys = reverse xs ++ ys` by induction. We will use this auxiliary result to show our main equality: `rev l = reverse l`.

```
reverse l
    = r l []                        // According to the definition of reverse.
    = rev l ++ []                   // Using the auxiliary result.
    = rev l                         // Using l ++ [] = l for any list l.          □
```

Actually the proofs presented above are not complete. Also the "obvious" properties of the operators should be proven to make the proofs complete. This is the topic of the exercises to come.

After you have found the way to prove a property, it is not difficult to do. Nevertheless, proving function correct is much work. This is the reason that it is seldom done in practise, despite the advantages of the increased confidence in the correctness of the program.

Program synthesis

In the previous section we treated the development of function definitions and proving their equivalence as two separate activities. In program synthesis these actions are integrated. We start with a specification, usually a naive and obviously correct implementation, and syntheses a new function definition. The reasoning required in both methods is essentially equal. In program synthesis we try to construct the function in a systematic way. Without synthesis these programs should be created out of thin air. In practise we often need to create the key step for program synthesis out of thin air. This key step is usually exactly equivalent to the step needed to construct the proof afterwards.

In program synthesis we use only a very limited number of transformations: *rewrite* according to a function definition (also called unfold [Burstall 87?]), *introduction of patterns and guards* (similar to the cases in proofs), *inverse rewrite steps* (this replacing of an expression by a function definition is called fold), and finally the introduction of *eureka definitions*.

In order to demonstrate program synthesis we will construct a recursive definition for `reverse` from the definition using the toolbox function `foldl`. The definition for `foldl` is repeated here to since it will be used in the transformation.

```
reverse :: [t] -> [t]
reverse l = foldl (\xs x -> [x:xs]) [] l          // The specification

foldl :: (a -> (b -> a)) a [b] -> a               // Toolbox function from StdEnv
foldl f r []    = r
foldl f r [a:x] = foldl f (f r a) x
```

The first step is to introduce patterns for the argument list. This is necessary since no rewrite step can be done without assumption on the form of the argument.

Case `l == []`

```
reverse l                                         // The function to transform
    = foldl (\xs x -> [x:xs]) [] l                // Using the specification.
    = foldl (\xs x -> [x:xs]) [] []               // The assumption of this case.
    = []                                          // Using alternative 2 of foldl.
```

Case `l == [a:x]`

```
reverse l                                         // Again the function to transform.
    = foldl (\xs x -> [x:xs]) [] l                // Using the specification.
    = foldl (\xs x -> [x:xs]) [] [a:x]            // Assumption l == [a:x].
    = foldl f (f [] a) x where f = \xs x -> [x:xs] // Alternative 1 of foldl.
    = foldl (\xs x -> [x:xs]) [a] x               // Lambda reduction.
    = foldl (\xs x -> [x:xs]) ([]++[a]) x         // Properties of ++.
    = foldl (\xs x -> [x:xs]) [] x ++ [a]         // Function foldl and λ-term.
    = reverse x ++ [a]                            // Fold to call of reverse.
```

Collecting the cases we have obtained:

```
reverse :: [t] -> [t]
reverse []    = []
reverse [a:x] = reverse x ++ [a]
```

In order to turn this in an algorithm that is linear in the length of the list we need the eureka definition `revAcc xs ys = reverse xs ++ ys`. Note that this is exactly equivalent to the key step of the proof in the previous section.

```
reverse l                                         // Function to transform.
    = reverse l ++ []                             // Property of ++.
    = revAcc l []                                 // Eureka definition.
```

To obtain a definition for `r` we use again pattern introduction:

Case `l == []`

```
revAcc [] ys = reverse [] ++ ys                   // Eureka definition and l == [].
    = [] ++ ys                                    // Definition of reverse.
    = ys                                          // Definition of operator ++.
```

Case `l == [a:x]`

```
revAcc [a:x] ys = reverse [a:x] ++ ys             // Eureka definition, l==[a:x].
    = reverse x ++ [a] ++ ys                      // Recursive definition reverse.
    = reverse x ++ [a:ys]                         // Associativity of ++.
    = revAcc x [a:ys]                             // Fold using eureka definition.
```

Collecting the cases we obtain:

```
reverse :: [t] -> [t]
reverse l = revAcc l []

revAcc :: [t] [t] -> [t]
revAcc [] ys    = ys
revAcc [a:x] ys = revAcc x [a:ys]
```

Since we used exactly the same eureka definition as was used in the proof, it is not surprising to see that we have obtain an completely equivalent definition of `reverse`. The key step in this kind of program synthesis is to discover the proper eureka definitions.

When you use uniqueness information in your functions you have to pay additional attention to the transformations. It is not sufficient that the uniqueness properties are preserved, but the compiler must also be able to verify them. Transformations using the first three steps are can in principle be automated to a certain extend [Wadler 88, Koopman ??] and will perhaps be incorporated into function language implementations in the future.

6.4 Tracing program execution

Despite of the careful design and incremental construction it might occur that you are completely puzzled by the behaviour of a function or program you have just written. The best strategy is to study all functions of your program individually. It might be useful to test individual functions with the tool described in chapter 5.

When this fail you might want to make a trace of the things happening in your program. Generally this can require a substantial redesign of your program since you have to pass a file or list around in your program in order to accommodate the trace. Fortunately, there is one exception to the environment passing of files. The file `stderr` can always be opened for writing errors. The trace can be modelled as a list of strings written to `stderr`.

As example we show how to construct a trace of the simple Fibonacci function:

```
fib n = let! y = fwrites ("fib "+toString n+" ") stderr in
        K (if (n<2) 1 (fib (n-1) + fib (n-2))) y

Start = fib 4
```

This yields the following trace:

```
fib 4 fib 2 fib 0 fib 1 fib 3 fib 1 fib 2 fib 0 fib 1
```

We usually write this trace as:

```
Start
→   fib 4
→   fib 3 + fib 2
→   fib 3 + fib 1 + fib 0
→   fib 3 + fib 1 + 1
→   fib 3 + 1 + 1
→   fib 3 + 2
→   fib 2 + fib 1 + 2
→   fib 2 + 1 + 2
→   fib 1 + fib 0 + 1 + 2
→   fib 1 + 1 + 1 + 2
→   1 + 1 + 1 + 2
→   5
```

From this trace it is clear that the operator + evaluates its second argument first.

It is tempting to write a function `trace :: !String x -> x` that writes the string as trace to `stderr`. The definition of this function is somewhat more complicated as you might expect:

```
trace :: !String x -> x
trace s x
 #! y   = fwrites s stderr
 | 1<2 = K x y
 = abort "?"
```

The problem with a simple minded approach like:

```
trace s x = let! y = fwrites s stderr in K x y
```

is that the strictness analyser of Clean discovers that `x` is always needed. So, `x` is evaluated before the function `trace` is invoked. This will spoil the order of the trace information. Switching strictness analysis off prevents this, but can change the order of evaluation in the program you are investigating.

Using the function `trace` we can write a version of the Fibonaccy function that produces a trace as:

```
fib n = trace   ("fib "+toString n+" ")
                (if (n<2) 1 (fib (n-1) + fib (n-2)))
```

When we want to include also the result of reduction is the trace we have to be vary careful that the order of computations is not changed. For some programs changing the order of computations is not a real problem. For other programs, changing the order of reductions can cause non-termination.

When we write:

```
fib n = trace ("fib "+toString n+" = "+toString m+" ") m
    where m = if (n<2) 1 (fib (n-1) + fib (n-2))
```

the trace will be reversed!

6.5 Higher order functions on lists

In the previous chapters we have seen some example of the use of higher order functions manipulating lists. Now you will be used to functional programming and probably able to appreciate these functions. In this section we will show some examples. Moreover, we will use the techniques introduced above to decide which foldl is most suitable in a given situation.

6.5.1 Displaying a number as a list of characters

The function `intChars` converts a positive number into a list of characters that contains the digits of that number. For example: `intChars 5678` gives the list `['5678']`. Thanks to this function you can combine the result of a computation with a list of characters, for example as in `intChars (3*17)++[' lines']`.

The function `intChars` can be constructed by the execution of a number of functions after each other. Firstly, the number should be repeatedly divided by `10` using `iterate` (like in the third example of `iterate` above). The infinite tail of zeroes is not interesting and can be chopped off by `takeWhile`. Now the desired digits can be found as the last digits of the numbers in the list; the last digit of a number is equal to the remainder after division by `10`. The digits are still in the wrong order, but that can be resolved by `reverse`. Finally the digits (of type `Int`) must be converted to the corresponding digit characters (of type `Char`). For this purpose we have to define the function `digitChar`:

```
digitChar :: Int -> Char
digitChar n
    | 0 <= n && n <= 9  = toChar (n + toInt '0')
```

An example clarifies this all:

```
            5678
             ↓   iterate (\x -> x / 10)
   [5678,567,56,5,0,0,…
             ↓   takeWhile (\y -> y <> 0)
     [5678,567,56,5]
             ↓   map (\z -> z rem 10)
        [8,7,6,5]
             ↓   reverse
        [5,6,7,8]
             ↓   map digitChar
    ['5','6','7','8']
```

The function `intChars` can now be simply written as the composition of these five steps. Note that the functions are written down in reversed order, because the function composition operator (`o`) means `after':

```
intChars :: (Int -> [Char])
intChars =  map digitChar
            o reverse
            o map (\z -> z rem 10)
            o takeWhile (\y -> y <> 0)
            o iterate (\x -> x / 10)
```

Functional programming is programming with functions!

Of course it is also possible to write a recursive function that does the same job. Actually, the function shown here works also for negative numbers and zero.

```
intToChars :: Int -> [Char]
intToChars 0 = ['0']
intToChars n | n<0  = ['-':intToChars (~n)]
             | n<10 = [digitChar n]
                    = intToChars (n/10) ++ [digitChar (n rem 10)]
```

6.5.2 Folding

There are several variants of the fold function. In this section we will compare them and give some hints on their use.

foldr

The `foldr` function inserts an operator between all elements of a list starting at the right hand with a given value. Cons is replaced by the given operator and nil by the supplied value:

```
                 xs  =   [   1   : [ 2   : [ 3   : [ 4   : [ 5   :   []  ]]]]]
                             ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓    ↓
      foldr (+) 0 xs  →  (   1   + ( 2   + ( 3   + ( 4   + ( 5   +    0  )))))
```

The definition in the standard environment is semantically equivalent to:

```
foldr :: (a->b->b) b [a] -> b
foldr op e []       = e
foldr op e [x:xs]   = op x (foldr op e xs)
```

By using standard functions extensively the recursion in other functions can be hidden. The `dirty work' is then dealt with by the standard functions and the other functions look neater. The function `or`, which checks whether a list of Booleans contains at least one value `True`, is, for example, defined in this way:

```
or :: ([Bool] -> Bool)
or = foldr (||) False
```

A lot of functions can be written as a combination of a call to `foldr` and to `map`. A good example is the function `isMember`:

```
isMember e  = foldr (||) False o map ((==)e)
```

In the same spirit we can define:

```
and :: ([Bool] -> Bool)
and = foldr (&&) True

sum :: ([t] -> t) | +, zero t
sum = foldr (+) zero

product :: ([t] -> t) | *, one t
product = foldr (*) one
```

The fold functions can also yield recursive datatypes. Some applications yielding a list are insertion sort and reverse:

```
isort :: ([t] -> [t]) | Ord t
isort = foldr insert []

reverser :: ([t] -> [t])
reverser = foldr (\x r -> r++[x]) []
```

The transformation of list to trees used in tree sort can also be written using a fold. This is an example of the use of fold yielding a recursive data type different from list.

```
listToTree :: ([t] -> Tree t) | Ord , Eq t
listToTree = foldr insertTree leaf
```

The fold functions are in fact very general. It is possible to write map and filter as applications of fold:

```
mymap :: (a -> b) [a] -> [b]
mymap f list = foldr ((\h t -> [h:t]) o f) [] list

mymap2 :: (a -> b) [a] -> [b]
mymap2 f list = foldr (\h t -> [f h:t]) [] list

myfilter :: (a -> Bool) [a] -> [a]
myfilter f list = foldr (\h t -> if (f h) [h:t] t) [] list
```

As a matter of fact, it is rather hard to find list manipulating functions that cannot be written as an application of fold.

foldl

The function `foldr` puts an operator between all elements of a list and starts with this at the end of the list. The function `foldl` does the same thing, but starts at the beginning of the list. Just as `foldr`, `foldl` has an extra parameter that represents the result for the empty list.

Here is an example of `foldl` on a list with five elements:

```
                  xs  =       [   1   : [ 2   : [ 3   : [ 4   : [ 5   : []]]]]
                                  ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
       foldl (+) 0 xs  =  (((((0 + 1)  +   2)  +   3)  +   4)  +   5)
```

The definition can be written like this:

```
foldl::(a -> (b -> a)) !a ![b] -> a
foldl op e []       = e
foldl op e [x:xs]   = foldl op (op e x) xs
```

The element `e` has been made strict in order to serve as a proper accumulator.

In the case of associative operators like `+` it doesn't matter that much whether you use `foldr` or `foldl`. Of course, for non-associative operators like `-` the result depends on which function you use. In fact, the functions `or`, `and`, `sum` and `product` can also be defined using `foldl`.

From the types of the functions `foldl` and `foldr` you can see that they are more general then the examples shown above suggest. In fact nearly every list processing function can be expressed as a fold over the list. As examples we show the identity function on lists, two versions of map, reverse and the function filter:

```
reversel :: [a] -> [a]
reversel l = foldl (\r x -> [x:r]) [] l
```

These examples are not intended to enforce you to write each and every list manipulation as a fold, they are just intended to show you the possibilities.

Folding to the right or to the left

Many functions can be written as a fold to the left, `foldl`, or a fold to the right, `foldr`. As we have seen in section 1, there are differences in the efficiency. For functions like `sum` it is more efficient to use `foldl`. The argument `e` behaves as an accumulator.

A function like reverse can be written using `foldl` and using `foldr`:

```
reversel l = foldl (\r x -> [x:r]) [] l

reverser l = foldr (\x r -> r++[x]) [] l
```

Difference in efficiency depends on the length of the list. The function `reverser` requires a number of reduction steps proportional to the square of the length of the list. For `reversel` the number of reduction steps is proportional to the length of the list. For a list of some hundreds of elements the difference in speed is about two orders of magnitude!

Can we conclude from these example that it is always better to use `foldl`? No, life is not that easy. As a counter example we consider the following definitions:

```
el = foldl (&&) True (repeat 100 False)
```

```
er = foldr (&&) True (repeat 100 False)
```

When we evaluate `el` the accumulator will become `False` after inspection of the first Boolean in the list. When you consider the behaviour of `&&` it is clear that the result of the entire expression will be `False`. Nevertheless, you program will apply the operator `&&` to all other Booleans in the list.

However, we can avoid this by using `foldr`. This is illustrated by the following trace:

```
foldr (&&) True (repeat 100 False)
→   foldr (&&) True [False: repeat (100-1) False]
→   (&&) False (foldr (&&) True (repeat (100-1) False))
→   False
```

That does make a difference! As a rule of thumb you should use `foldl` for operators that are strict in both arguments. For operators that are only strict in their first argument `foldr` is a better choice. For functions like reverse there is not a single operator that can be used with `foldl` and `foldr`. In this situation the choice should be determined by the complexity of the function given as argument to the fold. The function `\r x -> [x:r]` requires a single reduction step. While the function `\x r -> r++[x]` takes a number of reduction proportional to the length of x.

It requires some practise to be able to write functions using higher order list manipulations like fold, map and filter. It takes some additional training to appreciate this kind of definitions. The advantage of using these functions is that it can make the recursive structure of the list processing clear. The drawbacks are the experienced needed as a human to read and write these definitions and some computational overhead.

6.4 Exercises

1 In the function `qsort` there are two list comprehensions used to split the input list. It is possible to split the input list in one pass of the input list similar to `msort4`. Using an additional function it is possible to do this in one pass of the input list. Determine whether this increases the efficiency of the quick sort function.

2 To achieve the best of both worlds in the quick sort function you can combine splitting the input in one pass and continuation passing. Determine whether the combination of these optimisations does increase the efficiency.

3 When there are list elements with the same value it may be worthwhile to split the input list of the quick sort function in three parts: one part less than the first element, the second part equal to the first element and finally all elements greater than the first element. Implement this function and determine whether it increase the speed of sorting the random list. We can increase the amount of duplicates by appending the same list a number of times.

4 Determine and compare the runtime of the sorting functions `msort4`, `qsort4` (from the previous exercise), `tsort2` and `isort` for non-random lists. Use a sorted list and its reversed version as input of the sorting functions to determine execution times. Determine which sorting algorithm is the best.

5 Investigate whether it is usefull to pass the length of the list to sort as additional argument to merge sort. This length needs to be computed only once by counting the elements. In recursive calls it can be computed in a single reduction step. we can give the function `split` a single argument as accumulator. Does this increase the increase the efficiency?

6 Determine the time complexity of the functions `qsort2` and `tsort2`.

7 Study the behaviour of sorting functions for sorted lists and inversely sorted lists.

8 Proof that `abs (sign x) < 2` for all `x`. using:

```
sign x | x < 0  = -1
       | x == 0 = 0
       | x > 0  = 1
```

9 Prove that `fib n ≥ n` for all `n≥2`.

10 Prove that `l ++ [] = l`.

11 Prove that `x ++ (y ++ z) = (x ++ y) ++ z`. This is the associativity of `++`.

12 Prove that `rev (x ++ y) = rev y ++ rev x`.

13 Prove that `rev (rev xs) = xs` for every finite list.

14 Prove that `foldl (\xs x -> [x:xs]) ys x = foldl (\xs x -> [x:xs]) [] x ++ ys`.

15 Syntheses a recursive function to add elements of an list which is equivalent to:

```
sum :: [t] -> t | + , zero t
sum l = foldr (+) zero l
```

Design a Eureka rule to introduce an accumulator and transform the recursive function to a call of the addition function using the accumulator.