Functional Programs on Linear Structures

Christian Rinderknecht

Contents

Foreword

This book is intended for an undergraduate audience with a keen interest in discovering functional languages. It provides a slow-paced and deep delving into its subject, instead of covering as much field as possible with many different language features and applications. It is more akin to a field trip in geology rather than to a geographical relation. Accordingly, Erlang has been chosen amongst functional languages to serve this purpose because its core syntax is regular and plain and hence helps the reader to focus on the concepts at stake, not because it enables concurrent programming, which is a full-fledged topic in itself. As a consequence, it is important to bear in mind that this book is not specifically meant as an introduction to Erlang, but rather to kernels of many functional languages, more precisely those which enjoy a purely functional, strict and single-threaded semantics. In my experience, the lack of static type inference in Erlang is helpful in teaching beginners to write very short, first-order programs because they do not need to understand cryptic error messages when the program may actually work and when it does not, they quickly understand that something is wrong at run-time, without serious consequences. Also, not having to worry about user-defined error messages in the design, by means of incomplete definitions, allows the beginner to focus only on the correct behaviour on correct input. Another point of interest is that function definitions in an Erlang module are potentially mutually recursive, which allows us to introduce definitions in the most suitable order. Of course, in a more advanced course, complex data structures, robustness and module systems are best taught with a language featuring strong, static type inference.

This book is not structured as most textbooks are, in the sense that it contains overarching threads, intertwined in spirals like the trajectory of a space probe orbiting a planet until, having accumulated enough thrust, it reaches the orbit of another celestial body. This dynamics reflects on the contents as linear structures are the main focus of this introductory exposition but trees are graphically present all along; for example, I make early use of abstract syntax trees which are displayed in many figures. I hope that this graphical support will prove a complement to the traditional textual explanations.

The unusual length of this book is the unyielding consequence of its approach based on abundant examples, graphical illustrations, step by step resolutions and, furthermore, constant attempts at improving upon prior solutions. Also, in order to provide a unidirectional reading, references to other parts of the book are usually followed by a copy of the material referred to.

Some metaprogramming techniques are refined along, mainly transforming definitions into tail form. Such a focus is unusual in undergraduate textbooks but I deemed it relevant for two reasons. First, it helps infusing the idea that programs are data, since they too can be transformed, and allows dovetailing postgraduate textbooks. Second, it hopefully contributes clearing the countless misunderstandings students who worry about efficiency can read on the internet about definitions in tail form. Another peculiarity of this book lies in the execution model proposed for Erlang programs being based on abstract syntax trees. Control contexts, the control stack, the heap, garbage collection, aliasing and tail form optimisation are explained in terms of a memory model solely containing abstract syntax trees and references. A presentation more faithful to a modern interpreter would have required many details mostly relevant to the study of compilers and would have obscured the topics under focus.

Perhaps the last striking feature of this book is its conspicuous obsession with precision, as found in its numerous mathematical reckonings, equations and logical derivations. I indeed chose not to separate the writing of small programs from the study of their running time, which can sometimes be accurately predicted by a rigorous analysis. Not systematically developing such analyses often leads the students, in my experience, to neglect the concern of efficiency when designing algorithms. Moreover, it fails to underlie one of the advantages of the functional paradigm, which is to enable mathematical investigations of program properties as a complement to the usual methods of software engineering, like testing. The analytical and algebraic tools are almost always within the grasp of a motivated secondary school pupil. When not, perhaps when presenting examples of structural induction, I invite the reader to consider the challenge as a hook for future postgraduate studies.

It is advised not to skip any section and to perform a sequential reading from the start. If some part is found enigmatic upon first reading or overly long, just skim it. Often, new concepts are introduced informally or in passing and developed later, refined little by little, as it is normal in lecture notes. Some exercises are proposed and partially answered at the end of this book. These incomplete solutions are the last encouragement to complete the task.

I would like to express my gratitude towards Didier, who introduced me to Erlang and to my colleague HyungSeok Kim for suggesting me to write this book based on my lectures at Konkuk University, in South Korea. François Pottier (French National Institute for Research in Computer Science and Control, also known as INRIA), Philippe Flajolet (INRIA), Francisco Javier Barón López (Universidad de Málaga, Spain), Jaap Scherphuis and Nathaniel Johnston helped me with some mathematical derivations.

Seoul, Republic of Korea,
25th December 2023.
Ch. Rinderknecht

Chapter 1

Introduction

Consider the factorial function, defined by the following equations:

$$\begin{aligned} 1! &= 1, \\ n! &= n \cdot (n-1)!, \quad \text{if } n > 1. \end{aligned}$$

The expression '$n!$' reads as 'the factorial of the positive integer n,' where the factorial is a mathematical function symbolised by '!'. The equations mean that the *function call* $n!$ is the product of the consecutive positive integers up to n, which can be written informally as

$$n! = n \cdot (n-1) \cdot \ldots \cdot 1.$$

Let us use the equational definition to compute $3!$ as an example:

$$3! = 3 \cdot (3-1)! = 3 \cdot 2! = 3 \cdot (2 \cdot (2-1)!) = 3 \cdot (2 \cdot 1!) = 3 \cdot (2 \cdot (1)) = 6.$$

Let us be more precise about how this simple computation is carried out by tagging the equations with Greek letters:

$$\begin{aligned} 1! &\overset{\alpha}{=} 1, \\ n! &\overset{\beta}{=} n \cdot (n-1)!, \quad \text{if } n > 1. \end{aligned}$$

The previous computation is rewritten from left to right as follows:

$$3! \overset{\beta}{=} 3 \cdot (3-1)! = 3 \cdot 2! \overset{\beta}{=} 3 \cdot (2 \cdot (2-1)!) = 3 \cdot (2 \cdot 1!) \overset{\alpha}{=} 3 \cdot (2 \cdot (1)) = 6.$$

Two things are worth noticing. Firstly, not all equality symbols ($=$) are annotated. The unadorned ones correspond to arithmetic operations, like computing $3 - 1 = 2$, and thus are not part of the definition of the factorial function per se. Secondly, when a function call is to be replaced by the right-hand side of an equation, it is sometimes first surrounded by parentheses, like $2!$ being replaced by $\big(2 \cdot (2-1)!\big)$ instead of simply $2 \cdot (2-1)!$. The purpose is to avoid any interference with the *control context* of the call, that is, the part of the formula in which the call is embedded. In the current example, there is no ambiguity because the control context is always '$n \cdot ␣$', where the underscore (␣) is a placeholder for the call in question, but let us recall why parentheses are necessary in $x \cdot (y + z)$. The first step, that is, $3! \overset{\beta}{=} 3 \cdot (3 - 1)!$, does not require extra parentheses because there was no control context around 3! since it is the original call.

Follow the arrows. The way the previous equations were used suggests another notation, which provides further hints about their usage:

`fact(1)` $\xrightarrow{\alpha}$ `1;`
`fact(`n`)` $\xrightarrow{\beta}$ n `*`
`fact(`$n-1$`),` if $n > 1$.

We replaced the factorial function *symbol* (!) by the function *name* `fact`. Function names are placed before the value they apply to, for example, `fact(`n`)` instead of $n!$ which is usual in mathematics. Equality symbols have been replaced by arrows ($\rightarrow$) oriented from left to right. It is a fundamental property that equations are not oriented: every time $x = y$ is true, $y = x$ is as well and vice versa. In practice, when using equations to define a function, they are often implicitly oriented from left to right in order to hint at how they should be used to compute. But, in theory, nothing impedes writing

$$1! = 1,$$
$$n \cdot (n-1)! = n!, \quad \text{if } n > 1.$$

By replacing (=) with ($\rightarrow$), an orientation is imposed on the equations, which become *rewrite rules*. By restricting the way an equation is used, that is, either only from left to right or only from right to left, some generality may be lost, that is, some formulas might not be computed anymore, but it is a necessary compromise to automate computations. It is sometimes possible to complete rewrite rules with more rules in order to regain the lost expressivity when orienting the initial equations, but this is far off-topic here and we shall focus on designing rewrite rules from the start, instead of deriving them from a set of equations.

When considering rewrite rules, we must distinguish the left-hand side from the right-hand side of ($\rightarrow$). For example, the expressions used as input to a function in the right-hand side are called *arguments*, for instance, $n - 1$ is an argument of `fact`. The left-hand sides are restricted to be function calls and their arguments are either constants, like 1, or *variables*, which are names denoting *values*. Values are expressions which cannot be computed further, like an integer. These variables on the left-hand side are called *parameters*. So the n in `fact(`n`)` is a parameter of `fact` and n in 'n `*` ␣' is simply called a variable. Parameters refer potentially to an infinite number of values, for example, the parameter n in `fact(`n`)` can be bound to any integer. The previous computation is now written as follows:

$$
\begin{aligned}
\texttt{fact(3)} &\xrightarrow{\beta} \texttt{3 * fact(3-1)} &&= \texttt{3 * fact(2)}\\
&\xrightarrow{\beta} \texttt{3 * (2 * fact(2-1))} &&= \texttt{3 * (2 * fact(1))}\\
&\xrightarrow{\alpha} \texttt{3 * (2 * (1))} &&= \texttt{6}.
\end{aligned}
$$

The final integer is the value of the initial function call. A value contains neither function calls nor arithmetic operators, because such calls would need themselves to be thoroughly rewritten. For the moment, values can only be integers. It is often clearer to implicitly compose intermediary arithmetic operations $(=)$ with the current rewrite and write in short

$$\texttt{fact(3)} \xrightarrow{\beta} \texttt{3*fact(2)} \xrightarrow{\beta} \texttt{3*(2*fact(1))} \xrightarrow{\alpha} \texttt{3*(2*(1))} = \texttt{6}.$$

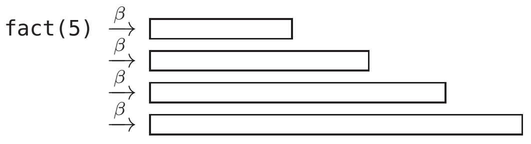
Note how the last rewrite $(\xrightarrow{\alpha})$ must be followed by a series of multiplications `3*(2*1)` because each individual multiplication had to be delayed until `fact(1)` be computed. This could have been anticipated because the call to `fact` in the right-hand side of the rewrite rule $(\xrightarrow{\beta})$, that is, the underlined text in

$$\texttt{fact(}n\texttt{)} \xrightarrow{\beta} n\ \texttt{*}\ \underline{\texttt{fact(}n\texttt{-1)}}$$

has the non-empty control context ‘n `*` ␣’. To understand why this is important, let us consider a slightly longer series of rewrites:

$$
\begin{aligned}
\texttt{fact(5)} &\xrightarrow{\beta} \texttt{5 * fact(4)}\\
&\xrightarrow{\beta} \texttt{5 * (4 * fact(3))}\\
&\xrightarrow{\beta} \texttt{5 * (4 * (3 * fact(2)))}\\
&\xrightarrow{\beta} \texttt{5 * (4 * (3 * (2 * fact(1))))}\\
&\xrightarrow{\alpha} \texttt{5 * (4 * (3 * (2 * (1))))} \quad = \texttt{120}.
\end{aligned}
$$

It is clear that each rewrite by $(\xrightarrow{\beta})$ yields a longer expression. An expression can contain function calls, arithmetic operations, numerical constants and variables. Therefore, *values are expressions but the converse does not always hold.* Let us focus now only on the shapes of the previous computation:



This phenomenon suggests that a great deal of space, that is, computer *memory*, is needed to keep the expressions before the final, long arithmetic computations. The example leads to induce that the larger term occurring in the computing of `fact(`n`)` is the one just before $(\xrightarrow{\alpha})$

$$\overbrace{\underbrace{\texttt{fact(N)}}_{\text{head}}\ \underbrace{\textbf{when}\ \texttt{N > 1}}_{\text{guard}}\ \texttt{->}\ \underbrace{\texttt{N * fact(N-1)}}_{\text{body}}}^{\text{clause}}$$

FIGURE 1: Structure of a clause in Erlang

is applied and its size is likely to be proportional to n, since all the integers from n to 1 had to be kept until the end.

The dual concept of space is time. By orienting the original equations, the definition becomes more operational, it conveys the notions of ‘before’ a rewrite, ‘after’ a rewrite and, transitively, of a ‘series’ of rewrites as a historical record. Therefore, the number of rewrites could be used as a measure of the time needed to compute the value of a function call. For instance, `fact(`n`)` requires n rewrites by $(\xrightarrow{\beta})$ plus one by $(\xrightarrow{\alpha})$, hence a total of $n + 1$.

Introducing Erlang. Erlang is a *functional programming language*, which means that it supports a programming style mainly based on the design and processing of mathematical functions. In Erlang, the previous rewrite system is written

```
fact(1)           -> 1;
fact(N) when N > 1 -> N * fact(N-1).
```

Notice how the syntax is similar to mathematics. A difference is that, in Erlang, the first letter of a variable must be set in upper case, like `N`, `Number` or `Var`, to distinguish it from a function name, whose first letter must be set in lower case, like `plusOne`. Let us remark that, mimicking a usage in English prose, each rewrite rule is either followed by a semicolon or a period, the entire definition being conceived as a complete sentence. Also, `when` is a *keyword*, that is to say, it looks like a function name but it is in practice reserved only for expressing conditions on a rewrite rule, like `N > 1`. In Erlang, a rewrite rule is called a *clause*. Its left-hand side is called the *head* and the right-hand side the *body*. A condition on a clause is called a *guard*. (As a side note, the lexical conventions, syntax and vocabulary of Erlang have been drawn indirectly from the Prolog programming language.) The structure of a clause in Erlang is summed up in FIGURE 1. A head has always the shape of a function call. A body is an expression. The guard in the definition of `fact` implies that there is no overlap between the two clauses. By overlap, we mean that, for a given function call, either no clause apply, thus no rewrite is entailed, or only one does. It is thus impossible for two clauses to carry on the next calculation. This becomes clearer if the clauses are rewritten in the following way:

```
fact(N) when N = 1 -> N;
fact(N) when N > 1 -> N * fact(N-1).
```

Obviously, the two guards are mutually exclusive, that is, when one is true the other one is false, thus enabling exactly one clause or none if `N < 1`. As a consequence, the previous definition could have been written instead as follows:

```
fact(N) when N > 1 -> N * fact(N-1);
fact(1)            -> 1.
```

The pieces of source code up to now are not complete Erlang programs for an Erlang program to be self-contained needs to be a *module*. A module is a unit of compilation containing a collection of function definitions. The module name must be the basename of the file containing the program. For example, if the definition above is found in a file named `math1.erl`, the corresponding module name must be `math1`:

```
-module(math1).                    % Drop the file extension .erl
-export([fact/1]).
fact(1)            -> 1;
fact(N) when N > 1 -> N * fact(N-1).
```

The `-export` line lists the function names which can be called from outside the module, that is, either from another module or from the Erlang *shell*. A shell is an application which reads commands entered by some user, interprets them, prints a result or an error message and waits for further commands. The ‘`/1`’ part of ‘`fact/1`’ means that the function named `fact` which has exactly one parameter is exported. The reason for specifying the number of parameters is that Erlang functions can be overloaded, that is, it is possible to define two functions with the same name as long as they differ in the number of their parameters.

In order to test some examples with `fact/1`, we first have to launch the Erlang shell. Depending on your operating system, your programming environment may vary greatly. Here, we shall assume a command-line interface, like the ones available in a terminal for the Unix-like operating systems. The Erlang shell is an application which allows us to interactively compile and call functions from modules and its name is likely to be `erl`. Here is the start of a session with the shell:

```
$ erl
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2]
[async-threads:0] [kernel-poll:false]
```

```
Eshell V5.6.3  (abort with ^G)
1> ▯
```

The first line is the command to run the shell. The last line is the prompt of the Erlang shell, the number meaning that the shell is waiting for the first command. Note that the operating system prompt is denoted by a dollar sign (`$`). The character ▯ denotes the blinking prompt of the Erlang shell where typing will occur. If we want to close the shell and return to the operating system shell, just type '`q().`' (standing for 'quit'). Each command must be terminated by a period (`.`) and followed by a pressure on the return key.

```
1> q().
ok
2> $ ␣
```

The character ␣ represents where text is to be typed in the operating system shell. But before quitting the Erlang shell, the first action usually consists in calling the Erlang compiler to process some module we want to use. This is done by the command '`c`', whose argument is the module name. In our example, the filename is `math1.erl`:

```
1> c(math1).
{ok,math1}
2> ▯
```

The compilation was successful, as the `ok` suggests. Let us compute some factorials now:

```
2> math1:fact(4).
24
3> math1:fact(-3).
** exception error: no function clause matching
math1:fact(-3)
4> ▯
```

The error message is very legible. In this book, we will rarely copy and paste the input to and the output from the Erlang shell. We will not write complete modules as well because we want to focus on the programming itself and delegate the practical aspects to a user manual or a textbook oriented towards practice.

A classic variation. Consider another way of writing the factorial:

```
-module(math2).
-export([fact_tf/1]).
```

```
fact_tf(N) when N >= 1 -> fact_tf(N,1).
fact_tf(1,A)           -> A;
fact_tf(N,A)           -> fact_tf(N-1,A*N).
```

The Erlang comparison (>=) stands for ‘greater or equal,’ which is usually written (=>) in other programming languages. This is because the sequence of symbols (=>) reminded too much of the logical implication ($\Rightarrow$). Notice how there are actually two definitions, each one concluded by a period. Both functions are named `fact_tf` but the first one takes only one parameter (the first clause), whilst the second one takes two parameters (the two last clauses). In other words, the former is completely qualified by `fact_tf/1` and the latter by `fact_tf/2`. We can remark also that Erlang allows us to write the definition of a function, such as `fact_tf/2`, *after* another definition making use of it, such as `fact_tf/1`. The function `fact_tf/2` is not meant to be called from outside the module, the rationale being that, instead of delaying the multiplications, exactly one multiplication is going to be computed at each rewrite, thus, in the end, nothing remains to be done—there is no instance of the control context to resume. The reason for this new version is hence to minimise the total memory needed to calculate the factorial. In order to follow an example with this new module `math2`, let us revert for a moment to the mathematical notation, so the clauses can be labelled with Greek letters and easily distinguished, but let us keep the Erlang variables:

$$
\begin{aligned}
&\texttt{fact\_tf(N)} &&\xrightarrow{\alpha} \texttt{fact\_tf(N,1)}, &&\text{if } \texttt{N > 0}.\\
&\texttt{fact\_tf(1,A)} &&\xrightarrow{\beta} \texttt{A;}\\
&\texttt{fact\_tf(N,A)} &&\xrightarrow{\gamma} \texttt{fact\_tf(N-1,A*N).}
\end{aligned}
$$

The previously considered function call `fact_tf(5)` is rewritten as

$$
\begin{aligned}
\texttt{fact\_tf(5)} &\xrightarrow{\alpha} \texttt{fact\_tf(5,1)}, && \text{since } \texttt{5 > 1},\\
&\xrightarrow{\gamma} \texttt{fact\_tf(5-1,1*5)} &&= \texttt{fact\_tf(4,5)}\\
&\xrightarrow{\gamma} \texttt{fact\_tf(4-1,5*4)} &&= \texttt{fact\_tf(3,20)}\\
&\xrightarrow{\gamma} \texttt{fact\_tf(3-1,20*3)} &&= \texttt{fact\_tf(2,60)}\\
&\xrightarrow{\gamma} \texttt{fact\_tf(2-1,60*2)} &&= \texttt{fact\_tf(1,120)}\\
&\xrightarrow{\beta} \texttt{120}.
\end{aligned}
$$

The reason why `fact_tf(5)` $=$ `fact(5)` is that

$$(((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2 = 5 \cdot (4 \cdot (3 \cdot (2 \cdot 1))). \tag{1.1}$$

This equality holds because, in general, for all numbers x, y and z,

1. the multiplication is associative: $x \cdot (y \cdot z) = (x \cdot y) \cdot z$;
2. the number 1 is neutral with respect to $(\cdot)$: $x \cdot 1 = 1 \cdot x = x$.

To show exactly why, let us write ($\overset{1}{=}$) and ($\overset{2}{=}$) to denote, respectively, the use of associativity and neutrality, then lay out the following equalities leading from the left-hand side to the right-hand side of the purported equality (1.1):

$$\begin{aligned}(((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2 &\overset{2}{=} ((((1 \cdot 5) \cdot 4) \cdot 3) \cdot 2) \cdot 1\\ &\overset{1}{=} (((1 \cdot 5) \cdot 4) \cdot 3) \cdot (2 \cdot 1)\\ &\overset{1}{=} ((1 \cdot 5) \cdot 4) \cdot (3 \cdot (2 \cdot 1))\\ &\overset{1}{=} (1 \cdot 5) \cdot (4 \cdot (3 \cdot (2 \cdot 1))\\ &\overset{1}{=} 1 \cdot (5 \cdot (4 \cdot (3 \cdot (2 \cdot 1))))\\ &\overset{2}{=} 5 \cdot (4 \cdot (3 \cdot (2 \cdot 1))). \quad \text{QED.}\end{aligned}$$

Furthermore, if we do not want to rely upon the neutrality of 1, we could set the initial call to `fact_tf(N-1,N)` and stop when the number is 0, instead of 1 (change in bold):

$$\begin{array}{lll}\texttt{fact\_tf(N) when N > 0} & \xrightarrow{\alpha} & \texttt{fact\_tf(N}\mathtt{\mathbf{-1}}\texttt{,}\mathtt{\mathbf{N}}\texttt{).}\\ \texttt{fact\_tf(}\mathtt{\mathbf{0}}\texttt{,A)} & \xrightarrow{\beta} & \texttt{A;}\\ \texttt{fact\_tf(N,A)} & \xrightarrow{\gamma} & \texttt{fact\_tf(N-1,A*N).}\end{array}$$

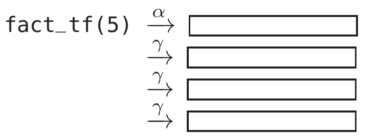
The same example now runs as

$$\begin{array}{llll}\texttt{fact\_tf(5)} & \xrightarrow{\alpha} \texttt{fact\_tf(5-1,5)} & = & \texttt{fact\_tf(4, 5)}\\ & \xrightarrow{\gamma} \texttt{fact\_tf(4-1,5*4)} & = & \texttt{fact\_tf(3, 20)}\\ & \xrightarrow{\gamma} \texttt{fact\_tf(3-1,20*3)} & = & \texttt{fact\_tf(2, 60)}\\ & \xrightarrow{\gamma} \texttt{fact\_tf(2-1,60*2)} & = & \texttt{fact\_tf(1,120)}\\ & \xrightarrow{\gamma} \texttt{fact\_tf(1-1,120*1)} & = & \texttt{fact\_tf(0,120)}\\ & \xrightarrow{\beta} \texttt{120}. & & \end{array}$$

This new version relies on the following equality which can be proved only by means of associativity:

$$(((5 \cdot 4) \cdot 3) \cdot 2) \cdot 1 = 5 \cdot (4 \cdot (3 \cdot (2 \cdot 1))).$$

The number of rewrites seems greater than with the factorial defined in module `math1`, precisely one more step due to the clause α. But this version presents an advantage in terms of memory usage, as long as it is understood that all integers within a certain range imposed by the hardware architecture occupy the same space at run-time. This means that we assume, for instance, that the memory needed to store the number `120` is the same as for the number `5`. Then the shape of the previous rewrites is



$$\xrightarrow{\gamma} \boxed{}$$
$$\xrightarrow{\beta} \boxed{}.$$

It is safe to induce that this version uses a constant chunk of memory, while the version in module `math1` uses an increasing amount of memory, more precisely a space proportional to `N` when computing `fact_tf(N)`. This phenomenon has been anticipated by the keen reader who noticed that there is no control context for the calls in the clauses of module `math2`, so there are no delayed computations that accumulate until the last step. As a conclusion, module `math2` is always preferable to `math1`.

Let us dwell a bit longer on the former. The relative order of the clauses β and γ is significant because the two corresponding heads can overlap, that is, there exists a *substitution* for the parameters which leads to identical heads. In this case, by substituting `0` for `N`, the heads of clauses β and γ become identical. Which one should be used when computing `fact_tf(0,A)`? The intention is to use clause β, but how can this preference be expressed? The problem could be remedied by adding a condition to clause γ:

`fact_tf(0,A)` $\xrightarrow{\beta}$ `A;`
`fact_tf(N,A)` $\xrightarrow{\gamma}$ `fact_tf(N-1,A*N)`, if `N` $\neq$ `0`.

In Erlang, this is expressed as a guard:

```
fact_tf(0,A)              -> A;
fact_tf(N,A) when N =/= 0 -> fact_tf(N-1,A*N).
```

Clearly, the two clauses now can be considered irrespectively of their relative order, so the following is just fine:

```
fact_tf(N,A) when N =/= 0 -> fact_tf(N-1,A*N);
fact_tf(0,A)              -> A.
```

With the aim to avoid writing too many guards, Erlang assumes an implicit order on the clauses as they are written. As a result, when looking for a match between a function call and a head in a clause, the firstly written clause is considered first, then, in case of match failure, the secondly written clause is examined etc. That is why

```
fact_tf(0,A) -> A;
fact_tf(N,A) -> fact_tf(N-1,A*N).
```

is valid because it is implicit that, in the second clause, `N` cannot be `0`, since the first clause would have been selected otherwise. Also, this means that the following definition is wrong.

`fact_tf(N,A)` $\xrightarrow{\gamma}$ `fact_tf(N-1,A*N);`
`fact_tf(0,A)` $\xrightarrow{\beta}$ `A.`

Indeed, the second clause would never be chosen: `N` in the first head can always be replaced by any number, including `0`. As a consequence, the computation never ends. For example:

$$\begin{aligned}\texttt{fact\_tf(3)} &\xrightarrow{\alpha} \texttt{fact\_tf(3,1)} \xrightarrow{\gamma} \texttt{fact\_tf(2,3)}\\ &\xrightarrow{\gamma} \texttt{fact\_tf(1,6)} \xrightarrow{\gamma} \texttt{fact\_tf(0,0)}\\ &\xrightarrow{\gamma} \texttt{fact\_tf(-1,0)} \xrightarrow{\gamma} \texttt{fact\_tf(-2,0)} \rightarrow \texttt{...}\end{aligned}$$

In other words, the Erlang definition ‘loops’ forever and the process running it has to be interrupted. The memory of the computer will not be exhausted because we already know that this version of factorial, albeit incorrect, uses a constant amount of space.

Still, would not it be better to guard `fact_tf(N,A)` from a negative `N`? For instance, by writing

```
fact_tf(N)   when N >= 1 -> fact_tf(N-1,N).
fact_tf(0,A)             -> A;
fact_tf(N,A) when N > 0  -> fact_tf(N-1,A*N).
```

Indeed, this is a more robust function `fact_tf/2`, since it now fails on incorrect input, that is, on negative or zero `N`, instead of entering an infinite loop. But note how the guard `N > 0` is evaluated each time the head of `fact_tf(N,A)` is successfully matched, which incurs a time penalty—even though we only take into account the number of rewrites to approximate the real time. Therefore, the only scenario where the guard is useful is when calling directly `fact_tf/2` without passing through the body of `fact_tf/1`. Since the only aim of `fact_tf/2` is to serve as an auxiliary to `fact_tf/1`, it could be envisaged that the guard be left off, while making sure that no direct calls to `fact_tf/2` are made in the rest of the module. Typically, a comment would remind the programmer of this requisite:

```
fact_tf(N) when N > 1 -> fact_tf(N-1,N).

% fact_tf(N,A) assumes N > 0.
fact_tf(0,A) -> A;
fact_tf(N,A) -> fact_tf(N-1,A*N).
```

Fibonacci. Consider the Fibonacci function defined as follows:

$$\begin{aligned}&\texttt{fib(0)} && \xrightarrow{\alpha} \texttt{1;}\\ &\texttt{fib(1)} && \xrightarrow{\beta} \texttt{1;}\\ &\texttt{fib(N) when N > 1} && \xrightarrow{\gamma} \texttt{fib(N-1) + fib(N-2).}\end{aligned}$$

Beware that some authors define `fib(0) -> 0`, leading to the sequence to be shifted by one: 0,1,1,2,3,5 etc. instead of our 1,1,2,3,5 etc. The peculiarity here is that the right-hand side of clause γ contains two function calls instead of one, as in the factorial function definition. This yields a cascade of choices as to which call is computed next. For instance, we underline the call to be rewritten in the rewrites of `fib(3)`:

$$
\begin{aligned}
\underline{\texttt{fib(3)}} &\xrightarrow{\gamma} \texttt{}\underline{\texttt{fib(2)}}\texttt{ + fib(1)}\\
&\xrightarrow{\gamma} \textbf{\texttt{(}}\texttt{fib(1) + }\underline{\texttt{fib(0)}}\textbf{\texttt{)}}\texttt{ + fib(1)}\\
&\xrightarrow{\alpha} \texttt{(fib(1) + }\textbf{\texttt{(1)}}\texttt{) + }\underline{\texttt{fib(1)}}\\
&\xrightarrow{\beta} \texttt{(}\underline{\texttt{fib(1)}}\texttt{ + (1)) + }\textbf{\texttt{(1)}}\\
&\xrightarrow{\beta} \texttt{(}\textbf{\texttt{(1)}}\texttt{ + (1)) + (1) = (2) + (1) = 3,}
\end{aligned}
$$

but the following rewrites are also admissible:

$$
\begin{aligned}
\underline{\texttt{fib(3)}} &\xrightarrow{\gamma} \texttt{fib(2) + }\underline{\texttt{fib(1)}}\\
&\xrightarrow{\beta} \texttt{}\underline{\texttt{fib(2)}}\texttt{ + (1)}\\
&\xrightarrow{\gamma} \texttt{(}\underline{\texttt{fib(1)}}\texttt{ + fib(0)) + (1)}\\
&\xrightarrow{\beta} \texttt{((1) + }\underline{\texttt{fib(0)}}\texttt{) + (1)}\\
&\xrightarrow{\alpha} \texttt{((1) + (1)) + (1) = (2) + (1) = 3.}
\end{aligned}
$$

The strategy of Erlang does not specify which call is rewritten first. The only constraint is that, for a call to be rewritten, its arguments must be values, that is, they do not contain calls (including arithmetic operations) themselves. We shall encounter later the case where function calls are arguments to other function calls and see what Erlang does in this case. In the previous example, the two original calls `fib(2)` and `fib(1)` fulfill this requirement. Even if the order of rewriting is not revealed to the programmer, it is guaranteed to be irrelevant. Moreover, the order is always the same for each execution of the program and only one call is computed at each step: these are the *deterministic* and *sequential* paradigms, respectively. Indeed, it may be imaginable that two or more calls are computed at the same time, giving birth to the (implicit) *concurrent* paradigm of computation. Something similar to the fictitious

$$
\begin{aligned}
\underline{\texttt{fib(3)}} &\xrightarrow{\gamma} \texttt{}\underline{\texttt{fib(2)}}\texttt{ + }\underline{\texttt{fib(1)}}\\
&\xrightarrow{\gamma\|\beta} \texttt{(}\underline{\texttt{fib(1)}}\texttt{ + }\underline{\texttt{fib(0)}}\texttt{) + (1)}\\
&\xrightarrow{\beta\|\alpha} \texttt{((1) + (1)) + (1) = (2) + (1) = 3.}
\end{aligned}
$$

But this is not the way Erlang works by default.

A look again at the two rewrites above reveals that `fib(1)` is computed twice. In order to know whether this is a coincidence, it is worth trying a longer example and use the concurrent strategy to save space:

$$
\begin{aligned}
\underline{\texttt{fib(5)}} &\xrightarrow{\gamma} \texttt{}\underline{\texttt{fib(4)}}\texttt{ + }\underline{\texttt{fib(3)}}\\
&\xrightarrow{\gamma\|\gamma} \texttt{(}\underline{\texttt{fib(3)}}\texttt{ + }\underline{\texttt{fib(2)}}\texttt{) + (}\underline{\texttt{fib(2)}}\texttt{ + }\underline{\texttt{fib(1)}}\texttt{)}\\
&\xrightarrow{\gamma\|\gamma} \texttt{((}\underline{\texttt{fib(2)}}\texttt{ + }\underline{\texttt{fib(1)}}\texttt{) + (}\underline{\texttt{fib(1)}}\texttt{ + }\underline{\texttt{fib(0)}}\texttt{))}\\
&\phantom{\xrightarrow{\gamma\|\gamma}} \texttt{ + ((}\underline{\texttt{fib(1)}}\texttt{ + }\underline{\texttt{fib(0)}}\texttt{) + 1)}
\end{aligned}
$$

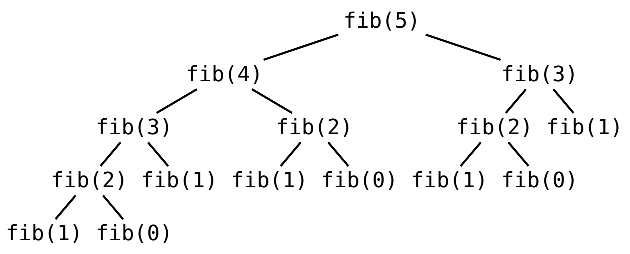


FIGURE 2: Call tree of `fact(5)`

$\longrightarrow$ ...

It is clear now that some computations are duplicated, leading to the following slow and memory-consuming Erlang program:

```
fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).                    % Assuming N > 1
```

In order to visualise the inefficiency, it is handy to use a two-dimensional representation of the calls such as the ones unfolding from `fib(5)` and shown in FIGURE 2. This kind of graphic is called a *tree* and FIGURE 3 shows its general shape. The disks are called *nodes* and the segments which connect two nodes are called *edges*. There are several kinds of nodes: the one at the top is called the *root* and the grey ones are called the *leaves*. Reading the tree top-down, a node is directly connected to some other nodes or none. If any, these nodes are called the *children* of the upper node and the latter is the *parent* of the formers. In our particular case, the tree is a *call tree* because each node represents a function call, the root being the initial call. Moreover, each node has two or zero children because each is an instance of a rewrite rule of the `fib` definition. For instance, clause γ corresponds to the tree pattern in

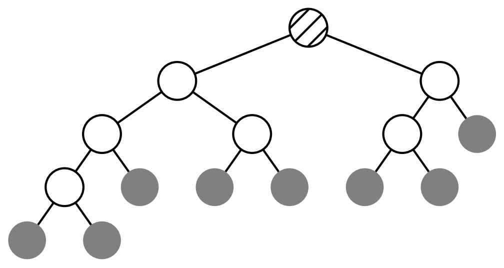


FIGURE 3: Shape of the call tree of `fib(5)` in FIGURE 2

FIGURE 4. It is now clear that some subtrees are repeated, the biggest one being rooted at the function calls `fib(3)`. Note also that the value of the root is the number or leaves, for example, the value of `fib(5)` is `8`. An interesting conclusion that can be drawn from this analysis is that implicit concurrency does not always result in an optimal computation, since some calls may be duplicated in parallel rewrites and, therefore, a better course of action is to find a faster version of the Fibonacci function within the sequential default strategy of Erlang.

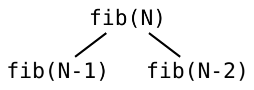


FIGURE 4: Generative tree pattern for the Fibonacci function

Linear structures. Until now, the only data type has been integers. In many situations, this is not versatile enough. Consider for example the need for summing a finite series of integers, like $11 + 5 + 7 + 3$. We must capture the concept of 'a finite series of values' in Erlang and make it amenable to computation. This is the very purpose of the *list* data structure. A list is either empty, in which case it is denoted in Erlang by the symbols `[]`, or non-empty. In the latter case, it is equivalent to say that it contains at least one *element*, also called *item*. Since a series is totally ordered, there must be a first item in a non-empty list. In Erlang, the first item is called the *head*—not to be confused with the head of a clause. The items following the head, if any, wholly constitute a list by themselves, called the *tail*. For instance, the list containing the item `1789` is written

```
[1789 | []].
```

Note the vertical bar which joins the head `1789` and the tail `[]`: it is an operator, that is, a predefined function denoted by a special symbol, called *push*, also called *cons*. The list containing items `1848` and `1789`, in this order, is specified as

```
[1848 | [1789 | []]].
```

The head is now `1848` and the tail is `[1789|[]]`. Variables can stand for any Erlang value, in particular, any item or tail. For instance,

```
[Head | Tail]
```

denotes a non-empty list. A list with at least two items would be written

```
[First | [Second | Tail]],
```

where `Tail` represents the tail of the tail of the whole list. Again here, Erlang follows the syntactic convention of the Prolog programming language. If a list, with or without variables, is used in the head of a clause, it is called a *pattern*. Actually, parameters and integers are patterns.

Summing. Consider now how to define a function `sum/1` summing all the integers in a given list. First, remember that a list is either empty or not, which leads to lay out the following canvas:

```
sum(   []) -> ▭;
sum([N|L]) -> ▭.
```

We have two patterns: `[]` (which is also a value) and `[N|L]`. The empty boxes are place holders for Erlang expressions to be determined. Firstly, it is worth noticing that the order of the clauses is not relevant here, since there cannot be any overlap between them (in other words, the two cases exclude each other), so

```
sum([N|L]) -> ▭;
sum(   []) -> ▭.
```

makes perfect sense as well. Thereafter, the programmer should focus separately on each body to be completed.

Let us take under examination the case of the empty list, for instance, and ponder what should be the outcome of summing no integers. It is very important to convince oneself that this matter does not make any sense and, in particular, the temptation of answering `0` must be vigorously dispelled. Natural numbers are often taught in primary school by repeatedly displaying heterogeneous collections of objects and stating that they contain the same ‘number’ of objects. The pupil is then expected to grasp the numerical concept by inductive reasoning. Of course, zero is naturally identified with the empty set because this pedagogy introduces the number as a cardinal. This conceptualisation of zero hinders the comprehension of algebra, where zero is abstractly and *positively* postulated as satisfying the equation $x + 0 = x$, for all numbers x (and assuming $x + y = y + x$). Formal thinking is essential to learn programming, therefore, we must convince ourselves that there is no natural relationship between zero and the empty list. Sometimes they can be associated, sometimes not: it all depends on the problem at hand.

Returning to the function `sum/1`, once it is understood that the empty list should not be accepted as an argument, the corresponding clause is simply removed and the definition boils down to

```
sum([N|L]) -> ▭.
```

In search for inspiration, turning to small and meaningful examples is often the way to go. For instance, what if we compute the function call `sum([5|[3|[7|[]]]])`? By comparing the shape of this call to the head of the only remaining clause, it should be apparent that if `5` is substituted for the variable `N` and `[3|[7|[]]]` is substituted for `L`, the head becomes identical to the function call. This is called a *match* and the whole process *matching*. It enables the clause to be used for computing the call. Since what is expected to be calculated by the body is $5+3+7$ or $3+5+7$ or $7+5+3$ etc., it seems natural to favour the combinations where the head `5` stands out, that is, either at the end of the sum or at the beginning: $5+\ldots$ or $\ldots+5$. Choosing arbitrarily the first form, the clause becomes

```
sum([N|L]) -> N + ☐.
```

It remains to understand the ellipsis in the formula. It stands either for $3+7$ or $7+3$. Is there a direct way to relate these additions to the tail `L`? In the example, the tail corresponds to `[3|[7|[]]]`. It should strike us as obvious that what remains to be done is exactly the same computation as previously but performed on the remaining numbers, that is, on the tail `L`. This is thus written:

```
sum([N|L]) -> N + sum(L).                    % A recursive call
```

This kind of clause, where the body contains a function call to the function being defined, while not being exceptional, is qualified by a special adjective in informatics: *recursive*. The corresponding definition, even if only one of its defining clause is recursive, is said to be recursive as well. Note that it is the definition that is recursive, not the function. Generally, a statement is recursive if it is self-referential. For instance, the definition of `fact_tf/2` in module `math2` is recursive. Once a definition is reached, it is wise to check it against another function call and unroll all the calls until a value is found:

```
sum([1|[2|[3|[]]]]) → 1 + sum([2|[3|[]]])
                    → 1 + (2 + sum([3|[]]))
                    → 1 + (2 + (3 + sum([])))
                    ↛
```

The broken arrow (↛) means that no clause is available to continue computing the function call `sum([])`. Indeed, the only clause at hand only accepts, that is ‘is matched by,’ non-empty lists. But the case of the empty list has been previously discarded as meaningless, so what should be done? Did we make a mistake?

While the sum of all the integers of the empty list indeed cannot be computed, the sum on the list containing *exactly one* integer can be, which means that the example above could simply terminate by means of an additional clause applied to `sum([3|[]])`:

```
sum([N| L]) -> N + sum(L);
sum([N|[]]) -> N.                     % New clause for singletons
```

First, we must understand that the variable `N` in the two clauses denote, in general, different values. In other words, the previous definition is exactly equivalent to

```
sum([N| L]) -> N + sum(L);
sum([M|[]]) -> M.                                     % Renaming
```

Second, the attentive reader has already noticed that the given complete definition still fails to compute the input above. The reason is that the head of the first clause is matched by all non-empty lists, that is, all lists with at least one item, in particular, the lists with exactly one item, called *singletons*. But the second clause, which has just been added, is also matched by singletons, so the heads of the two clauses overlap partially. Nevertheless, Erlang specifies that the first matching head will select the corresponding body, that is to say, heads are tried in turn until one is matched by the arguments of the function call to be computed. If none is matched, an error occurs at run-time, as symbolised by the broken arrow ($\nrightarrow$) above. In our example, the first clause is always matched until the empty list is considered again (as bound by `L`), which leads to the exact same failure as presented previously. The fix consists simply in swapping the clauses and write instead

```
sum([N|[]]) -> N;                               % Must come first
sum([N| L]) -> N + sum(L).
```

For the sake of legibility, let us label the arrows:

$$\begin{aligned}&\texttt{sum([N|[]])} \xrightarrow{\alpha} \texttt{N;}\\ &\texttt{sum([N| L])} \xrightarrow{\beta} \texttt{N + sum(L).}\end{aligned}$$

and revisit, now successfully, our running example:

$$\begin{aligned}\texttt{sum([1|[2|[3|[]]]])} &\xrightarrow{\beta} \texttt{1 + sum([2|[3|[[]]]])}\\ &\xrightarrow{\beta} \texttt{1 + (2 + sum([3|[]]))}\\ &\xrightarrow{\alpha} \texttt{1 + (2 + (3))} \qquad = \texttt{6.}\end{aligned}$$

As a side-note, it is nevertheless possible to write the second clause first if a guard is used:

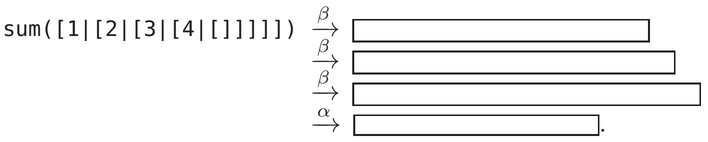
```
sum([N| L]) when L =/= [] -> N + sum(L);             % L ≠ []
sum([N|[]])               -> N.
```

but this version, although more explicit, is generally avoided by good programmers, who prefer concise definitions and always keep in mind the order Erlang applies to the clauses.

Efficiency. What can be said about the speed and the memory usage of the function `sum/1`? The number of rewrites clearly equals the number of integers in the list because every integer is either matched with one pattern head or the other. Hence, if the initial function is called on a list of n integers, the number of steps to reach the result is n: $n-1$ times using clause β and one time using clause α. Taking a slightly longer list can provide a hint about memory usage:

$$\begin{array}{ll}
\texttt{sum([1|[2|[3|[4|[]]]]])} & \xrightarrow{\beta} \texttt{1 + sum([2|[3|[4|[]]]])} \\
 & \xrightarrow{\beta} \texttt{1 + (2 + sum([3|[4|[]]]))} \\
 & \xrightarrow{\beta} \texttt{1 + (2 + (3 + sum([4|[]])))} \\
 & \xrightarrow{\alpha} \texttt{1 + (2 + (3 + (4)))} \qquad = \texttt{10}.
\end{array}$$

which prompts us to consider only the sizes of the right-hand sides:



It seems that the total memory usage increases slowly and then reduces sharply after the last rewrite step. But this calls for a closer examination. In particular, the character font used here is proportional, which implies that, for instance, (`+`) takes more horizontal space on the line than (`|`). Omitting blanks and keeping a monospace font gives

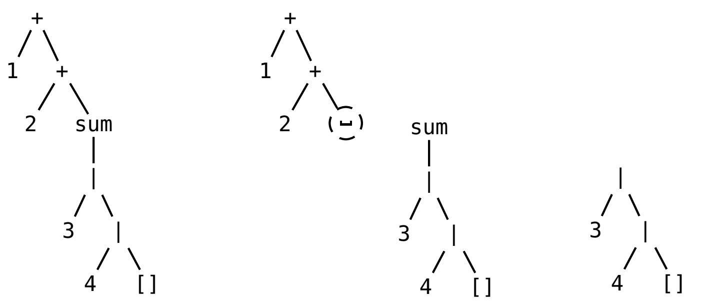
```
sum([1|[2|[3|[4|[]]]]]) -> 1+sum([2|[3|[4|[]]]])
                        -> 1+(2+sum([3|[4|[]]]))
                        -> 1+(2+(3+sum([4|[]])))
                        -> 1+(2+(3+(4))).
```

it seems now that the expressions are of constant size until clause α applies. Moreover, even if (`+`) were instead written `plus`, its occurrence should not be considered as taking more memory than (`+`) because names are only tags. Also, what about the parentheses and the blanks? Should they be considered meaningful, as far as memory allocation is concerned? All these considerations bring to the fore the need for a finer understanding of how Erlang functions and data are usually represented at run-time but, because these encodings depend strongly on the compiler and the hardware architecture, it would be inappropriate to rely on too detailed a description. Therefore, it is sufficient and appropriate here to provide a model of expressions which allows to draw conclusions about memory usage which hold up to a proportional constant, while reminding the reader that this model is unlikely implemented as it is presented next.

Memory. Up to now, the only elements composing expressions have been variables, integers, function calls, arithmetic operators and lists of expressions. (Incidentally, this is a recursive definition.) The key is to realise that the essence of an expression is best captured by a two-dimensional representation, as opposed to a line of punctuated text. This kind of model is widespread in mathematics as matrices, graphics, graphs etc. The best suited here is a kind of tree called *abstract syntax tree*. We introduced trees on page 18 in the context of the *call trees* of a function computing the Fibonacci numbers.

Consider in FIGURE 5a the abstract syntax tree of the expression `1+(2+sum([3|[4|[]]]))` and the function call it contains in FIGURE 5c. The leaves here are the integers `1`, `2`, `3` and `4` and the empty list `[]`. The inner nodes, that is, the ones which are not leaves, are either arithmetic operators, like (`+`), or function names, like `sum`, or the *list constructor* (`|`). By taking as origin the node `sum`, the abstract syntax tree can be split into the part below it, that is, the argument (see FIGURE 5d), and the part above, called *instances of the control context* (see FIGURE 5b).

The main interest of abstract syntax trees is that no parentheses are required, because a sub-expression is denoted by a subtree, that is, a tree embedded into another. Moreover, blank characters are absent as well. This allows us to focus on the essential. To illustrate the gained practicality, consider again the previous computation in full, from left to right (the node to be rewritten is boxed), in FIGURE 6 on the facing page. It is now clear that the instances of the control context accumulate so as to grow in inverse proportion to the argument's length: integers move one by one from the argument to the control context and the



(a) Expression (b) Instances of the control context `N + ␣` (c) Call (d) Argument

FIGURE 5: `1+(2+sum([3|[4|[]]]))`

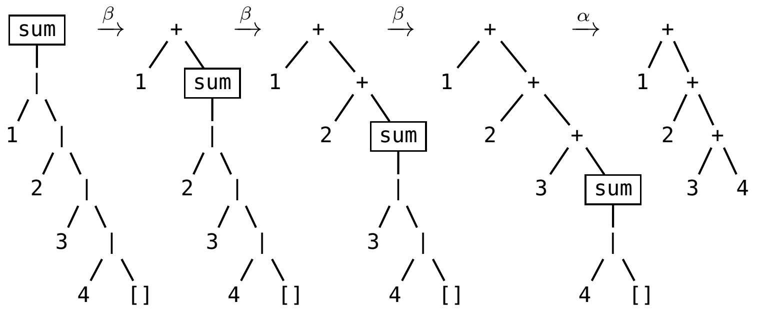


FIGURE 6: Computation of `sum([1,2,3,4])`

associated operation changes from a push (`|`) to an addition. Therefore, if we use as memory unit a node, the total memory is indeed constant—except for the last step.

The part of the memory where the instances of the control context accumulate is called the *call stack*. Since the instances have always the same shape (an addition), the size of the call stack can be thought of as proportional to the control context size, the unit of measure being a node. The original function call and its arguments are also stored in the call stack, except if the arguments are lists, in which case the call stack contains a *reference* to a different part of the memory, called the *heap*, where the lists are stored. The exact nature of this reference is not relevant here. Suffice to say that it is an oriented edge ending downwardly at a node (`|`) or `[]`. Consider again the running example and what remains in the call stack, step after step, in FIGURE 7. The arrow represents a reference to a list in the heap, as it is the unique argument of `sum/1` (the list is not shown). This example shows that the call stack grows until it contains the result or a reference to it if it is located in the heap, while the heap size will decrease so that the total memory, that is, the size of the call stack plus the heap size, remains

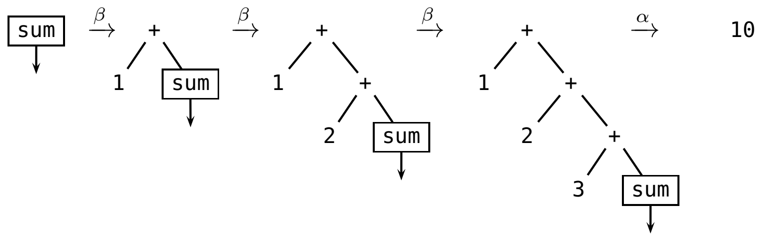
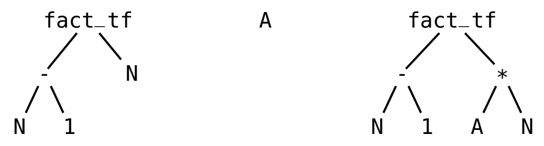


FIGURE 7: Call stack while computing `sum([1,2,3,4])`



(a) `fact_tf(N-1,N)` (b) `A` (c) `fact_tf(N-1,A*N)`

FIGURE 8: Abstract syntax trees from `fact_tf/1` and `fact_tf/2`

constant. In case the value of a call is a list, the call stack would contain a reference to the heap, where the value in question resides.

Tail form. On most operating systems, the maximum size of the call stack is often much smaller than the maximum heap size, therefore, if this is a concern, it is wise to write programs that do not overload the call stack. In the case of the function `sum/1`, if the input list contains n integers, the size of the call stack increases proportionally to n. Is there a way to write another definition of the function so that the heap is mainly used instead? The best is to try to express `sum/1` without using any control context at all: the size of the call stack will then be constant because, in this configuration, only the arguments will reside in the stack—the arguments which are lists are stored in the heap and a reference to them is kept in the call stack instead. This kind of function definition is said to be in *tail form*. Such a definition was presented before, on page 14, as a variation on the first factorial function:

```
fact_tf(N) when N > 0 -> fact_tf(N-1,N).
fact_tf(0,A)          -> A;
fact_tf(N,A)          -> fact_tf(N-1,A*N).
```

The abstract syntax trees denoting the three bodies are either a value (see FIGURE 8b) or a function call whose arguments contain no user-defined function call (see FIGURE8a and 8c). Arithmetic expressions `N-1` and `A*N` call *operators*, that is, built-in functions represented by symbols.

Is there a tail form definition of `sum/1`? First, let us call this new function `sum_tf/1` (*tail **f**orm*) in order to avoid any confusion. The idea, very much alike the factorial, is to define an auxiliary function `sum_-tf/2` with a supplementary argument which accumulates partial results. This kind of argument is called an *accumulator*. The new version should hence look like as follows:

```
sum_tf(L)         -> sum_tf(□,L).
sum_tf(A,[N|[]]) -> □;
sum_tf(A,[N| L]) -> □.
```

or, equivalently,

```
sum_tf(L)         -> sum_tf(L,□).
sum_tf([N|[]],A) -> ▭;
sum_tf([N| L],A) -> ▭.
```

Notice that, just as with `sum/1`, the call `sum_tf([])` fails without a warning and we may find this behaviour questionable. Indeed, it can be considered inappropriate in the framework of software engineering, where programming large and robust applications is a requisite, but this book focuses primarily on programming in the small, therefore the programs introduced here are purposefully fragile; in other words, they may fail on some undesirable inputs instead of providing the user with some nice warnings, error messages or, even better, managing to get the compiler itself to reject such programs.

Since it was decided that an accumulator is needed, we must be clear on what kind of data it holds. As said previously, an accumulator usually contains a part of the final result. From a different perspective, an accumulator can be regarded as a partial trace of all the previous rewrite steps. Here, since the final result is an integer, we bear in mind that the accumulator ought to be a number as well.

There is no need to fill the above canvas (the boxes) from the first line to the last: this is a program, not an essay. Perhaps the best method is to first lie down the heads of the clauses and make sure none is missing and that none is useless (taking into account the implicit ordering from first to last). Second, we pick up the clause whose body seems the easiest to guess. For instance, the first clause of `sum/2` seems simple enough because it applies when only one number, `N`, remains in the input list. Since the accumulator `A` holds the partial sum up to now, only `N` remains to be processed. Therefore, the answer is `N+A` or `A+N`:

```
sum_tf(L)         -> sum_tf(L,□).
sum_tf([N|[]],A) -> N+A;
sum_tf([N| L],A) -> ▭.
```

The second clause of `sum_tf/2` is chosen next. It applies when the input list is not empty and its first item is `N` and the remaining are in `L`. Until now, the partial sum is the accumulator `A`. It is clear that a recursive call is needed here, because if the body were `N+A` again, then the rest of the integers, `L`, would be useless. So the process must be resumed with another input:

```
sum_tf(L)         -> sum_tf(L,□).
sum_tf([N|[]],A) -> N+A;
sum_tf([N |L],A) -> sum_tf(□,▭).
```

The question now is to find what the new list and the new accumulator should be in this last clause. What is known about the list? `N` and `L`. What can be done with `N`? Well, the same that was done before, in the first clause of `sum_tf/2`, that is, let us add it to the accumulator:

```
sum_tf(L)        -> sum_tf(L,□).
sum_tf([N|[]],A) -> N+A;
sum_tf([N| L],A) -> sum_tf(□,N+A).
```

This way, the new accumulator is `N+A`, which is fine since the purpose of the accumulator is to hold the partial sum until the present number, which is `N` now. What new list of numbers should be used? It is clear that `N` cannot be reused here, because it has already been added to the accumulator, and it must not be added twice. This means that it is not needed anymore. Remains `L`, which is what is sought, since it represents all the remaining numbers to be added to the accumulator:

```
sum_tf(L)        -> sum_tf(L,□).
sum_tf([N|[]],A) -> N+A;
sum_tf([N| L],A) -> sum_tf(L,N+A).
```

The last unfinished business is the initial value of the accumulator. It is important not to rush and to deal with this value at the last moment. What kind of operation is being carried out on the accumulator? Additions. Without knowing anything about the integers in `L`, as it is the case in the clauses of `sum_tf/1`, what integer could be taken as an initial value? It is well known that, for all n, $0 + n = n + 0 = n$, thus `0` appears to be the only possible value here, because it does not change the total sum:

```
sum_tf(L)        -> sum_tf(L,0).
sum_tf([N|[]],A) -> N+A;
sum_tf([N| L],A) -> sum_tf(L,N+A).
```

The last step consists in trying some examples after the labelling

```
sum_tf(L)        $\xrightarrow{\alpha}$ sum_tf(L,0).
sum_tf([N|[]],A) $\xrightarrow{\beta}$ N+A;
sum_tf([N| L],A) $\xrightarrow{\gamma}$ sum_tf(L,N+A).
```

Consider our running example again:

```
sum_tf([1|[2|[3|[4|[]]]]])
  $\xrightarrow{\alpha}$ sum_tf([1|[2|[3|[4|[]]]]],0)
  $\xrightarrow{\gamma}$ sum_tf([2|[3|[4|[]]]],1+0) = sum_tf([2|[3|[4|[]]]],1)
  $\xrightarrow{\gamma}$ sum_tf([3|[4|[]]],2+1)     = sum_tf([3|[4|[]]],3)
  $\xrightarrow{\gamma}$ sum_tf([4|[]],3+3)         = sum_tf([4|[]],6)
  $\xrightarrow{\beta}$  4 + 6                      = 10.
```

Let us recall here the run

```
sum([1|[2|[3|[4|[]]]]]) -> 1+sum([2|[3|[4|[]]]])
                        -> 1+(2+sum([3|[4|[]]]))
                        -> 1+(2+(3+sum([4|[]])))
                        -> 1+(2+(3+(4))).
```

The difference between `sum_tf/1` and `sum/1` lies not in the result (both functions are indeed equivalent) but in the way the additions are performed. They are equivalent because

$$4 + (3 + (2 + (1 + 0))) = 1 + (2 + (3 + 4)).$$

This equality holds because, for all numbers x, y and z,

1. the addition is associative: $x + (y + z) = (x + y) + z$,
2. the addition is symmetric: $x + y = y + x$,
3. zero is a right-neutral number: $x + 0 = x$.

To show exactly why, let us write $(\overset{1}{=})$, $(\overset{2}{=})$ and $(\overset{3}{=})$ to denote, respectively, the use of associativity, symmetry and neutrality, and lay out the following equalities leading from one expression to the other:

$$\begin{aligned}
4 + (3 + (2 + (1 + 0))) &\overset{3}{=} 4 + (3 + (2 + 1))\\
&\overset{2}{=} (3 + (2 + 1)) + 4\\
&\overset{2}{=} ((2 + 1) + 3) + 4\\
&\overset{2}{=} ((1 + 2) + 3) + 4\\
&\overset{1}{=} (1 + 2) + (3 + 4)\\
&\overset{1}{=} 1 + (2 + (3 + 4)). \quad \text{QED.}
\end{aligned}$$

This seems a bit heavy for such a small program. Is there a way to rewrite further `sum_tf/1` so that less hypotheses are needed to prove the equivalence with `sum/1`? Let us start with the most obvious difference: the use of zero. This zero is the initial value of the accumulator and its sole purpose is to be added to the first number in the list. We could then simply first load the accumulator with this number, so the neutrality of zero is no more required:

```
sum_tf([N|L])    -> sum_tf(L,N).
sum_tf([N|[]],A) -> N+A;
sum_tf([N| L],A) -> sum_tf(L,N+A).
```

But this definition of `sum_tf/1` fails on lists containing exactly one number, because `L` can be empty. Therefore, we must allow the list to be empty in the definition of `sum_tf/2`:

```
sum_tf([N|L])   -> sum_tf(L,N).
sum_tf(   [],A) -> A;
sum_tf([N|L],A) -> sum_tf(L,N+A).
```

Now, we can easily get rid of the hypothesis that the addition is symmetric: by replacing `N+A` by `A+N`:

```
sum_tf([N|L])   -> sum_tf(L,N).
sum_tf(   [],A) -> A;
sum_tf([N|L],A) -> sum_tf(L,A+N).
```

Let us relabel the arrows

```
sum_tf([N|L])   $\xrightarrow{\alpha}$ sum_tf(L,N).
sum_tf(   [],A) $\xrightarrow{\beta}$ A;
sum_tf([N|L],A) $\xrightarrow{\gamma}$ sum_tf(L,A+N).
```

and consider again our running example:

```
sum_tf([1|[2|[3|[4|[]]]]])
         $\xrightarrow{\alpha}$ sum_tf([2|[3|[4|[]]]],1)
         $\xrightarrow{\gamma}$ sum_tf([3|[4|[]]],1+2) = sum_tf([3|[4|[]]],3)
         $\xrightarrow{\gamma}$ sum_tf([4|[]],3+3)     = sum_tf([4|[]],6)
         $\xrightarrow{\gamma}$ sum_tf([],4+6)         = sum_tf([],10)
         $\xrightarrow{\beta}$  10.
```

This time, the series of additions corresponds to the expression $((1+2)+3)+4$, which we can prove equal to $1+(2+(3+4))$ by means of the associativity of (+) only:

$$((1+2)+3)+4 \overset{1}{=} (1+2)+(3+4) \overset{1}{=} 1+(2+(3+4)). \quad \text{QED.}$$

What about the speed and the memory usage of `sum_tf/1`? While this is not an exact measure, let us assume that the speed is proportional to the number of rewrites. Then, it is easy to convince ourselves that each step by means of clauses β and γ process exactly one integer from the input list, so the total number of rewrite steps is the number of integers plus one due to the initial rewrite through clause α. In other words, if the initial input list contains n integers, the number of rewrites is exactly $n+1$. When n is large and taking into account the assumption we made about this measure of speed, this means that the execution time is proportional to the size of the input, for all inputs. In particular, if the input size (taken here as the length of the list) triples, then the running time also triples.

Consider the abstract syntax trees of the rewritten expressions in figure 9 on the next page, where the arrow ($\twoheadrightarrow$) means ‘at least one rewrite.’ Two expressions e_1 and e_2 are said to be equal, noted $e_1 \equiv e_2$, if $e_1 \twoheadrightarrow v$ and $e_2 \twoheadrightarrow v$. For instance, `sum([1,2,3,4])` $\equiv$ `sum([5,5])`. The intermediary trees where the abstract syntax tree of `A+N` appears have been skipped to show the point, which is that the size of the trees

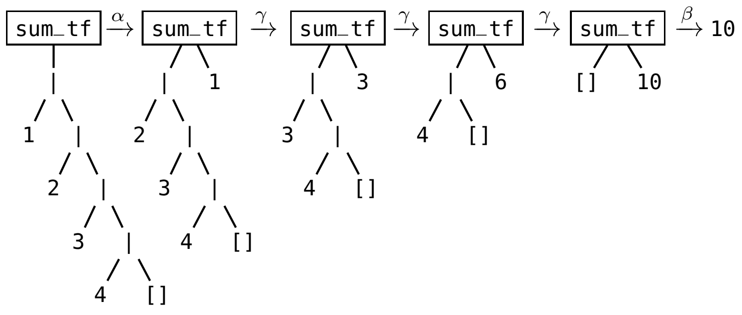


FIGURE 9: Abstract syntax trees of `sum_tf([1,2,3,4])` $\twoheadrightarrow$ `10`

strictly decreases and the size of the call stack is constant (the boxed nodes all remain at the first level). There is no control context to the recursive call of `sum_tf/2`, so no call stack space is needed for it, but the call stack is not empty because the initial function call to rewrite is always put in the call stack, including the arguments which are integers and the references to lists in the heap (but not the arguments which are lists). The parts of the trees which reside in the call stack at each step are shown in FIGURE 10. As mentionned earlier, the downward arrows

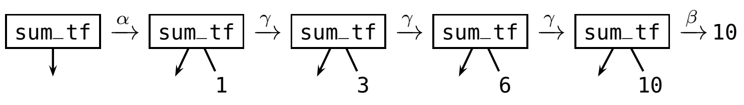


FIGURE 10: Call stack while computing `sum_tf([1,2,3,4])`

stand for references to values in the heap, where the first argument, being a list, is stored. It becomes clear now that the size of the call stack remains constant and the heap size decreases during the computation, hence so does the total memory size used by the program. As a conclusion, the `sum_tf/1` is to be preferred over `sum/1` because it consumes less call stack and heap usage also decreases during the calculation after the first rewrite.

Does this mean that `sum_tf/1` should always be used when summing integers?

Let us consider again the previous example. In mathematical notations, what is aimed at are sums of the type $S_n = 1 + 2 + \cdots + n$. By writing it from right to left, this is still $S_n = n + (n-1) + \cdots + 1$. Adding side by side these two equalities yields

$$S_n + S_n = (1 + n) + (2 + (n-1)) + \cdots + (n+1),$$

$$2 \cdot S_n = \underbrace{(n+1) + \ldots + (n+1)}_{n \text{ times}} = (n+1) \cdot n,$$

so $S_n = n(n+1)/2$. In particular, it comes without a surprise that $S_4 = 4 \cdot (4+1)/2 = 10$. In Erlang, this is the program

```
s(N) when N >= 0 -> N*(N+1)/2.
```

This function is very efficient, both in terms of speed and memory usage, since only one step, involving only arithmetic operators, is needed. The moral of this story is that sometimes a specialised version is more suitable.

Multiplying. Consider this time multiplying all the integers in a list. The first thing that should come to the mind is that this problem is very similar to the previous one, only the arithmetic operator is different, so the following definition can be written immediately, by modification of `sum/1`:

```
mult([N|[]]) -> N;
mult([N| L]) -> N * mult(L).
```

Similarly, a definition in tail form can be derived from `sum_tf/1`:

```
mult_tf([N|L])   -> mult_tf(L,N).
mult_tf(   [],A) -> A;
mult_tf([N|L],A) -> mult_tf(L,A*N).
```

This program inherits the constant call stack of `sum_tf/1`. Moreover, the reason why `mult_tf/1` is equivalent to `mult/1` is the same reason why `sum_tf/1` is equivalent to `sum/1`: the arithmetic operator (*) is associative, just as (+) is.

What could be improved that could not be in `sum_tf/1`? In other words, what can speed up a long series of multiplications? The occurrence of at least one zero, for example. In that case, it is not necessary to keep multiplying the remaining numbers, because the result is going to be zero anyway. This optimisation can be done by setting apart the case when `N` is `0`:

```
mult_tf([N|L])   -> mult_tf(L,N).
mult_tf(   [],A) -> A;
mult_tf([0|L],A) -> 0;                               % Improvement.
mult_tf([N|L],A) -> mult_tf(L,A*N).
```

How often a zero occurs in the input? In the *worst case*, there is no zero and thus the added clause is useless. But if it is known that zero is likely to be in the input with a probability higher than for other numbers, this added clause could be useful in the long term, that is, on the average time of different runs of the program. Actually, even if the numbers are uniformly random over an interval including zero, it makes sense to keep the clause.

Here have been reviewed different ways of assessing the computation time: one is to count the exact number of rewrite steps for any input; another is to find, when the input is large for some measure, an equivalent function, for example, saying that the running time t_n is proportional to the input of size n can be written $t_n \sim an$, as $n \to \infty$, for some positive constant a and measure n; another way consists in considering the running time when the input is large and has the topology that maximally slows down the execution (the worst case); another one is to consider the average running time for random inputs.

Chapter 2

Joining and reversing

Consider writing a function `join/2` with two lists as parameters and which computes a list containing all the items of the first list followed by all the items of the second one. In other words, all function calls `join(`P`,`Q`)` are rewritten into a list containing all the items of P followed by all the items of Q. For instance, the function call `join([3|[5|[]]],[2|[]])` has value `[3|[5|[2|[]]]]`. Since the above requirement is a bit vague, it is important to try out several extreme cases in order to understand more precisely what is expected from this function. These extreme cases are special configurations of the arguments. Both of them being lists, a hint comes naturally to mind because it is well known that lists must either be empty or non-empty. This simple observation leads to consider four distinct cases: both lists are empty; the first is empty and the second is not; the first is not empty and the second is; both are not empty:

```
join(   [],   [])
join(   [],[J|Q])
join([I|P],   [])
join([I|P],[J|Q])
```

It is important not to rush to write the bodies. Are some cases left out? No, because there are exactly two arguments which can each be either empty or not, leading to exactly $2 \cdot 2 = 4$ cases. Then, we write the corresponding Erlang canvas:

```
join(   [],   []) -> ▭;
join(   [],[J|Q]) -> ▭;
join([I|P],   []) -> ▭;
join([I|P],[J|Q]) -> ▭.
```

Let us ponder which clause looks easier, because there is no reason to complete them in order if do not want to. When reading the patterns, a clear mental representation of the situation should arise. Here, it seems that the first clause is the easiest: what is the list made of all the items of the (first) empty list followed by all the items of the (second) empty list? Since the empty list, by definition, contains no item, the answer is the empty list:

```
join(   [],   []) -> [];
join(   [],[J|Q]) -> □;
join([I|P],   []) -> □;
join([I|P],[J|Q]) -> □.
```

Perhaps we might wonder whether this case should be dropped. In other words, is it a meaningful case? Yes, because the vague English description of the behaviour of `join/2` mandates the result be always a list made of items of other lists (the arguments), so, in the absence of any items to “put” in the result, the resulting list “remains” empty.

It should be evident enough that the second and third clauses should lead to the same result, because when appending a non-empty list to the right of an empty list is the same as appending it to the left: the result is always the non-empty list in question.

```
join(   [],   []) -> [];
join(   [],[J|Q]) -> [J|Q];          % Clause symmetrical...
join([I|P],   []) -> [I|P];                 % ...to this one.
join([I|P],[J|Q]) -> □.
```

The last clause is the trickiest. What does the pattern in the clause head reveal about the situation? That both lists are not empty, more precisely, the head of the first list is denoted by the variable `I`, the corresponding tail (which can either be empty or not) is `P`, and similarly for the second list and `J` and `Q`. Are these bricks enough for building the body, that is, the next step towards the result? We understand that appending two lists P and Q preserves in the result the total order of the items in P and Q but also the relative order of the items of P with respect to the items of Q. In other words, the items of P must be present in the result before the items of Q and the items from P must be in the same order as in P (same for Q). With this ordering in mind, it is worth sketching the following

```
join([I|P],[J|Q]) -> □ I □ P □ J □ Q □.
```

In one rewrite step, can the item `I` be placed in its final position, that is, in a position from where it does not need to be moved again? The answer is yes: it must be the head of the result.

```
join([I|P],[J|Q]) -> [I | □ P □ J □ Q □].
```

What about the other head, variable `J`? It should remain the head of `Q`:

```
join([I|P],[J|Q]) -> [I | □ P □ [J | Q]].
```

What is the relationship between `P` and `[J|Q]`? We can be tempted by

```
join([I|P],[J|Q]) -> [I | [P | [J | Q]]].                % ?
```

which, despite being correct Erlang, is flawed. The reason is that, while Erlang allows lists to be used as items for other lists (in other words, lists can be arbitrarily embedded in other lists), `P` should not be used as an item here, as it is when it is used as a head for `[J|Q]`. Consider the running example `join([3|[5|[]]],[2|[]])`: here, the head of the clause in question bounds `I` to `3`, `P` to `[5|[]]`, `J` to `2` and `Q` to `[]`, therefore the corresponding putative body `[I|[P|[J|Q]]]` is actually `[3|[`**`[5|[]]`**`|[2|[]]]]`, which is different from the expected result `[3|[`**`5`**`|[2|[]]]]` in that `[5|[]]` is not `5`.

How can `P` and `[J|Q]` be joined? Well, is this question not familiar yet? There is a list, here named `P`, which has to be appended to the left of another list, whose shape is described by the pattern `[J|Q]`. Of course, this is exactly the purpose of the function `join/2` being defined. Therefore, what we need here is a recursive call:

```
join(   [],   []) -> [];
join(   [],[J|Q]) -> [J|Q];
join([I|P],   []) -> [I|P];
join([I|P],[J|Q]) -> [I|join(P,[J|Q])].
```

Even before proofing this idea against some examples, we should consider on the spot whether this kind of recursive definition is likely to terminate for all valid inputs. Indeed, while Erlang allows any kind of recursive definition, this does not mean that all lead to terminating programs. Let us consider

```
loop(N) -> loop(N).
```

which is a valid program but which does not end for any input and uses a constant amount of memory. (Some other ill-conceived programs run until they exhaust the memory of its corresponding process.) Hence we must discriminate between terminating and non-terminating recursive calls. There is no general automatic method which can always answer that question and that is why the Erlang compiler does not check the termination property. In practice, however, some heuristics work quite well, as we shall see. First, the pattern and the call must be compared for a match, and a measure of the arguments, for instance, the size in the case of a list, must be decided upon and kept in mind. In the case of `loop/1`, there is only one argument, which is an integer and the head of the clause is exactly the same as the body: `loop(N)`. It is not surprising that such definition never ends. But then what about the following?

```
loop(N) -> loop(N+1).
```

In this situation, the clause head is different from the recursive call. A rather “natural” measure on an integer is induced by the total ordering on the mathematical integers, that is, we rely upon `N < N + 1`. This implies that the recursive call is going to be performed on increasingly larger numbers without an end. Let us consider a slight variation:

```
loop(N) -> loop(N-1).
```

Here `N - 1 < N`, but that does not seem to make any difference as far as termination is concerned. But a clause can be added to *filter*, that is, catch, the terminating value of the argument. For instance

```
loop(0) -> 7;
loop(N) -> loop(N-1).
```

This is a program which always terminate with the value `7`, for example,

$$\texttt{loop(3)} \rightarrow \texttt{loop(2)} \rightarrow \texttt{loop(1)} \rightarrow \texttt{loop(0)} \rightarrow \texttt{7}.$$

Two conditions are sufficient to have a terminating recursive definition: the recursive calls must have smaller arguments than their heads and some clauses must provide pattern heads for the smallest arguments. But we may object that the variant definition

```
loop(0) -> 7;
loop(N) -> loop(N+1).
```

also terminates on `7` for all `N < 0`. This remark reveals by the way that

```
loop(0) -> 7;
loop(N) -> loop(N-1).
```

fails to terminate if `N < 0`, so it should have been written instead

```
loop(0)            -> 7;
loop(N) when N > 0 -> loop(N-1).
```

Returning to the other loop, the following terminates for all `N < 1`:

```
loop(0)            -> 7;
loop(N) when N < 0 -> loop(N+1).
```

These observations allow us to formulate more precisely the previous criteria: a recursive definition always terminates if the size of the arguments of the recursive calls change so that the series of calls gets strictly closer, that is, converges, to some cases which terminate in one step, called *base cases*. In the above definitions, the base case is `loop(0)` and the recursive calls bring one step closer to it.

If we apply this criterion to `join/2`, we can see that the first argument “becomes” a shorter list in one rewrite: `P` is shorter by one item in comparison with `[I|P]`, whilst the second argument does not change. Since the direction of change is towards a smaller first argument, base cases are needed to filter the minimum value of the first argument. The initial reasoning about the requirements revealed that the first argument can be the empty list and two complementary base cases are already written: `join([],[])` and `join([],[J|Q])`. Therefore, the function `join/2` terminates on all input lists.

By the way, an assumption was made earlier about the loop

```
loop(N) -> loop(N).
```

which was too strong. It was indeed implicitly taken for granted that `N` was an integer, but this statement does not arise from the examination of the code. Actually, `N` could perfectly well be a list, since nothing is done with it. The moral of this story is that we should not trust naming conventions when inferring the type of a variable, especially when inspecting the code of another programmer. It is not because a variable is spelled `N` that it is an integer, it is its uses in the program that make it an integer or not. Of course, this remark does not entail that no naming convention should be followed. In

```
loop(0)            -> 7;
loop(N) when N < 0 -> loop(N+1).
```

there are two constraints which entail that `N` must be an integer: the guard `N < 0` and the arithmetic operation `N+1`. Continuing on this track, the definition of `join/2` could be examined in terms of the nature, or types, of the items in the lists. Let us consider again the definition

```
join(   [],   []) -> [];
join(   [],[J|Q]) -> [J|Q];
join([I|P],   []) -> [I|P];
join([I|P],[J|Q]) -> [I|join(P,[J|Q])].
```

Do `I` and `J` have to be integers? By looking at the bodies, it appears that none of these variables are involved in any kind of computation or guard that would force them to denote numbers; they are passed around until they reach their respective position in the final result. Spurred on by curiosity, we should try a test with embedded lists, so let us label the arrows of the previous definition:

```
join(   [],   []) $\xrightarrow{\alpha}$ [];
join(   [],[J|Q]) $\xrightarrow{\beta}$ [J|Q];
join([I|P],   []) $\xrightarrow{\gamma}$ [I|P];
```

join([I|P],[J|Q]) $\xrightarrow{\delta}$ [I|join(P,[J|Q])].

and try for instance

join([3|[[4|[]]|[]]],[[]|[]])
$\xrightarrow{\delta}$ [3|join([[4|[]]|[]],[[]|[]])]
$\xrightarrow{\delta}$ [3|[[4|[]]|join([],[[]|[]])]]
$\xrightarrow{\beta}$ [3|[[4|[]]|[[]|[]]]].

It may be unclear what the arguments in this example are. The first list contains two items: the first one is the number 3 and the last one is the *list* containing the single number 4. The second argument is the list containing a single item, the empty list. Fortunately, Erlang proposes an alternative syntax which is more legible and the previous example is preferably written

join([3,[4]],[[]]) $\xrightarrow{\delta}$ [3|join([[4]],[[]])]
$\xrightarrow{\delta}$ [3,[4]|join([],[[]])]
$\xrightarrow{\beta}$ [3,[4],[]].

There are two simplifying conventions:

- for all expression e, we have [e|[]] $\equiv$ [e];
- for all expressions e_0 and e_1, and all lists L, we have

[e_0|[e_1|L]] $\equiv$ [e_0,e_1|L].

In particular, a list whose items are all known can be specified by enumerating all these items separated by commas, for example, the list [3|[1|[0|[]]]] is the same as [3,1,0] and [5|[]] is the same as [5]. About the second simplification, note that [3|[1|L]] is the same as [3,1|L], but [3|[1|L]] *is not* [3,1,L], because L is not an item of the whole list: it is the rest of the whole list following the two first items.

As a final note, it is easy to convince oneself that the function join/2 is *associative*, that is, for all lists L_1, L_2 and L_3, the following holds:

join(L_1,join(L_2,L_3)) $\equiv$ join(join(L_1,L_2),L_3).

Correctness and completeness. It is extremely common that, in the course of designing a function, we focus on some aspect, some clause, and then on some other part of the code, and we may miss the forest for the trees. When we settle for a function definition, the next step is to check whether it is correct and complete with respect to the conception we had of its expected behaviour.

We say that a definition is *correct* if all function calls we can make with it are rewritten by the clauses into the expected result *and* every call failing was expected to fail. By failure, we mean that a *stuck expression* is reached, that is, an expression which can not be further rewritten into a value. We came across such an expression on page 21 and represented this situation with a broken arrow ($\nrightarrow$).

We say that a definition is *complete* if all function calls that we expect to be computable are indeed computable. In other words, we must also check that the definition enables rewriting into values all the inputs we deem acceptable.

How do we check that the last definition of `join/2` is correct and complete? If the concept of “expected result” is not formally defined, typically by means of mathematics, we resort to *code review* and *testing*. One important aspect of the reviewing process consists in verifying again the heads of the definition and see if all possible inputs are accepted or not. In case some inputs are not matched by the heads, we must justify that fact and record the reason in a comment. The heads of `join/2` match all the combinations of two lists, whether they are empty or not, and this is exactly what was expected: no more, no less. The next step is to inspect the bodies and wonder twofold: (1) Are the bodies rewritten into the expected type of value, for all function calls? (2) Are the function calls being provided the expected type of arguments? These checks stem from the fact that Erlang does not include *type inference* at compile-time. Other functional languages, like OCaml and Haskell, would have their compilers automatically establish these properties. The examination of the bodies in the definition of `join/2` confirms that

- the bodies of the clauses α, β and γ are lists containing the same kind of items as the arguments;
- the arguments of the unique recursive call in clause δ are lists made of items from the parameters;
- assuming that the recursive call has the expected type, we deduce that the body of the last clause is a list made of items from the arguments.

As a conclusion, the two questions above have been positively answered. Notice how we had to assume that the recursive call already had the type we were trying to establish for the current definition. There is nothing wrong with this reasoning, called *inductive*, and it is rife in mathematics. We shall revisit it in different contexts.

The following stage consists in testing the definition. This means to define a set of inputs which lead to a set of outputs and failures that are all expected. For example, it is expected that `join([],[])` be rewritten into `[]`, so we could assess the validity of this statement by running the code. And the function call indeed passes the test. How should we choose the inputs meaningfully? There are no general rules, but some guidelines are useful. One is to consider the empty case or the smaller case, whatever that means in the context of the function. For example, if some argument is a list, then let us try the empty list. If some argument is a nonnegative integer, then let us try zero. Another advice is to have at least *test cases*, that is, some function calls whose values are known, which exert each clause. In the case of `join/2`, there are four clauses to be covered by the test cases.

Improving joins. When we are convinced that the function we just defined is correct and complete, it is often worth considering again the code for improvement. There are several directions in which improvements, often called *optimisations* although the result is not optimal, can be achieved:

- Can we rewrite the definition so that in all or some cases it is faster?
- Is there an equivalent definition which uses less memory in all or some cases?
- Can we shorten the definition by using less clauses (perhaps some are useless or redundant) or shorter bodies?
- Can we use less parameters in the definition? (This is related to memory usage.)
- Do we need a tail form version of the same function? (Again, this is related to memory, more precisely, the call stack.)

Let us reconsider `join/2`:

```
join(   [],   []) -> [];
join(   [],[J|Q]) -> [J|Q];
join([I|P],   []) -> [I|P];
join([I|P],[J|Q]) -> [I|join(P,[J|Q])].
```

and focus our attention on the two first clauses, whose common point is to have the first list being empty. It is clear now that the bodies, which are values, are, in both clauses, the second list, whether it is empty (first clause) or not (second clause). Therefore, there is no need to discriminate on the structure of the second list when the first one is empty and we can equivalently write

```
join(   [],    Q) -> Q;                          % Q is a list
join([I|P],   []) -> [I|P];
join([I|P],[J|Q]) -> [I|join(P,[J|Q])].
```

Note how the new definition does not ascertain that `Q` is a list—hence the comment—so it is not strictly equivalent to the original definition: now `join([],5)` $\rightarrow$ `5`. Let us compromise by favouring the conciseness of the latter definition.

Let us consider next the two last clauses and look for common patterns. It turns out that, in the penultimate clause, the first list is matched as [I|P] but nothing is done with I and P except *rebuilding* [I|P] in the body. This suggests that we could simplify the clause as follows:

```
join(   [],    Q) -> Q;
join(    P,   []) -> P;                   % P is a non-empty list
join([I|P],[J|Q]) -> [I|join(P,[J|Q])].
```

It is important to check that changing [I|P] into P does not affect the matching, that is to say, exactly the same inputs which used to match the head are still matching it. Indeed, it is possible in theory that the new P matches an empty list. Can we prove that P is always non-empty? The head of the penultimate clause matches only if the previous clause did not match, since Erlang tries the heads in the order of the writing, that is, top-down. Therefore, we know that P can not be the empty list, because [] is used in the previous clause *and* the second parameter can be any list. Nevertheless, as happened before, P is not necessarily a list anymore, for instance, join(5,[]) $\rightarrow$ 5. Again, we will ignore this side-effect and choose the conciseness of the latter definition.

In the last clause, we observe that the second argument, matched as [J|Q], is simply passed over to the recursive call, thus it is useless to distinguish J and Q and we can try

```
join(   [], Q) -> Q;
join(    P,[]) -> P;
join([I|P], Q) -> [I|join(P,Q)].                 % Can Q be []?
```

Again, we must make sure that Q can not match an empty list: it can not be empty because, otherwise, the previous head would have matched the call. As it is, the penultimate head is included in the last, that is, all input matching the last could match the penultimate, if we let aside the order used by Erlang, which leads us to consider whether the function would still be correct if Q could be empty after all. If so, then the penultimate clause would be useless. Let us label the clauses as follows:

```
join(   [], Q) $\xrightarrow{\alpha}$ Q;
join(    P,[]) $\xrightarrow{\beta}$ P;
join([I|P], Q) $\xrightarrow{\gamma}$ [I|join(P,Q)].
```

Let L be some list containing n items, which we write informally as $L = [I_0, I_1, \ldots, I_{n-1}]$. The subscript i in I_i is the *position* of the item in the list, the head of the list being at position 0. Then Erlang would rewrite in one step

$$\texttt{join}(L,\texttt{[]}) \xrightarrow{\beta} L.$$

Had clause β been erased, we would have had instead the series

$$\begin{aligned}
\texttt{join}(L,\texttt{[]}) &\xrightarrow{\gamma} \texttt{[}I_0\texttt{|join([}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],[])]}\\
&\xrightarrow{\gamma} \texttt{[}I_0\texttt{|[}I_1\texttt{|join([}I_2\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],[])]]}\\
&= \texttt{[}I_0\texttt{,}I_1\texttt{|join([}I_2\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],[])]}\\
&\xrightarrow{\gamma} \texttt{[}I_0\texttt{,}I_1\texttt{|[}I_2\texttt{|join([}I_3\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],[])]]}\\
&= \texttt{[}I_0\texttt{,}I_1\texttt{,}I_2\texttt{|join([}I_3\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],[])]}\\
&\;\vdots\\
&\xrightarrow{\gamma} \texttt{[}I_0\texttt{,}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{|join([],[])]}\\
&\xrightarrow{\alpha} \texttt{[}I_0\texttt{,}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{|[]]}\\
&= \texttt{[}I_0\texttt{,}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{]}\\
&:= L.
\end{aligned}$$

(The notation (:=) means “equal by definition,” sometimes called an *assignment*. It must be distinghished from (=), which is an equality *deduced* from some other property.) In short, we found

$$\texttt{join}(L,\texttt{[]}) \twoheadrightarrow L.$$

This means that clause β is useless, since its removal allows us to reach the same result L, although more slowly: n steps by clause γ plus 1 by clause α, instead of one step by clause β. We are hence in a situation where we discover that the original definition was already specialised for speed when the second list is empty. If we remove clause β, the program is shorter but becomes slower in that special case. This kind of dilemma is quite common in programming and there is sometimes no clear-cut answer as to what is the best design. Perhaps another argument can here tip the scale slightly in favour of the removal. Indeed, whilst the removal slows down some calls, it makes the number of rewrites easy to remember: it is the number of items of the first list plus 1; in particular, the length of the second argument is irrelevant. So let us settle for

```
join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

Let us note that `join(5,[])` fails again, as it does in the original version.

When programming medium or large applications, it is recommended to use evocative variables, like `ListOfProc`, instead of enigmatic ones, like `L`. But in this presentation we are mostly concerned with short programs, not software engineering, so short variables will do. Nevertheless, we need to opt for a naming convention so we can easily recognise the type of the variables across function definitions. Let us decide that `L`, `P`, `Q` and `R` are lists, while `I` and `J` denote elements, also called items, of those lists (which can be lists themselves).

Tail form. As the rewrite of `join(`L`,[])` shows, the definition of `join/2` is not in tail form. Indeed, a function call is embedded in the second clause and it has the control context `[I|␣]`, the mark `␣` standing for the location of the call. We may wonder whether a tail form variant is necessary or not. This decision lies entirely on practical grounds, typically, the maximum size of the call stack in comparison to the call stack usage as a function of the expected size of the input. In the case under consideration, the call stack usage can be considered as proportional to the size of the first list, as the list grows. More precisely, we ended up with a definition of `join/2` for which the number of computational steps is $n+1$ if the length of the first argument is n. Therefore, if we expect some calls to `join/2` to have a first argument longer than the maximum size of the call stack allocated by the process running the program, it is wise to write a tail form version of `join/2`, otherwise a crash is likely. But it is never necessary in theory, hence experience is paramount here. For the sake of the argument, let us imagine that we need such a tail form variant. Perhaps it is also useful here to state unambiguously that it is always possible to rewrite a function definition into an equivalent tail form definition.

For the moment, instead of presenting an automatic transformation, let us envisage an empirical approach. We previously considered the transformations of `sum/1` into `sum_tf/1`, on page 30, and of `mult/1` into `mult_tf/1`, on page 32. The idea consisted in adding an *accumulator* (or *accumulative parameter*) which would hold a partial result. This partial result was computed by applying the control context to the previous value of the accumulator, thus, for instance, the clause

```
sum([N|L]) -> N + sum(L).
```

became

```
sum_tf([N|L],A) -> sum_tf(L,A+N).
```

But we noted that this works only because the addition operation is associative, that is, $x + (y + z) = (x + y) + z$. In the case of `join/2`, as stated above, the control context is `[I|␣]` and the operator (`|`) is not associative: `[X|[Y|Z]]` $\not\equiv$ `[[X|Y]|Z]`. Let us try nevertheless and see exactly what happens. We want a new function `join/3` whose shape is

```
join(   [],Q,A) -> [                ];
join([I|P],Q,A) -> [                ].
```

The new parameter `A` is the accumulator in question. Since we want to push `I` on it, it must be a list. Furthermore, its initial value should be the empty list, otherwise extraneous items would be found in the result. Therefore, the definition in tail form, equivalent to `join/2` and named `join_tf/2`, calls `join/3` with the extra argument `[]`:

```
join_tf(P,Q) -> join(P,Q,[]).
```

Let us go back to `join/3` and push `I` on `A`:

$$\begin{array}{l}
\texttt{join(\ \ \ [],Q,A)} \xrightarrow{\alpha} \boxed{}\texttt{;}\\
\texttt{join([I|P],Q,A)} \xrightarrow{\beta} \mathbf{\texttt{join(P,Q,[I|A])}}\texttt{.}
\end{array}$$

What is the accumulator with respect to the expected result? We already know that it can not be a partial result, because (`|`) is not associative. So some more work has to be done with `A` *and* `Q`, but, first, we should understand what `A` contains at this point and unfolding a call, with a piece of paper and a pencil, is quite enlightening. Let L be a list of n items $[I_0, I_1, \ldots, I_{n-1}]$. We have the following:

$$\begin{array}{rl}
\texttt{join(}L\texttt{,}Q\texttt{,[])} & \xrightarrow{\beta} \texttt{join([}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],}Q\texttt{,[}I_0\texttt{])}\\
& \xrightarrow{\beta} \texttt{join([}I_2\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{],}Q\texttt{,[}I_1\texttt{,}I_0\texttt{])}\\
& \vdots\\
& \xrightarrow{\beta} \texttt{join([],}Q\texttt{,[}I_{n-1}\texttt{,}I_{n-2}\texttt{,}\ldots\texttt{,}I_0\texttt{])}\\
& \xrightarrow{\alpha} \boxed{}\texttt{.}
\end{array}$$

Therefore `A` in the head of clause α is bound to a list which contains the same items as the original first argument L, but in *reverse order*. In other words, given the call `join(P,Q,[])`, the parameter `A` in the first head of `join/3` holds `P` reversed. What can we do with `A` and `Q` in order to reach the result? The key is to realise that the answer depends on the contents of `A`, which, therefore, needs to be matched more accurately: is `A` empty or not? This leads to split clause α into α_0 and α_1:

$$\begin{array}{l}
\texttt{join(\ \ \ [],Q,\ \ \ }\mathbf{\texttt{[]}}\texttt{)} \xrightarrow{\alpha_0} \boxed{}\texttt{;}\\
\texttt{join(\ \ \ [],Q,}\mathbf{\texttt{[I|A]}}\texttt{)} \xrightarrow{\alpha_1} \boxed{}\texttt{;}\\
\texttt{join([I|P],Q,\ \ \ \ A)} \xrightarrow{\beta} \texttt{join(P,Q,[I|A]).}
\end{array}$$

Notice that clauses α_0 and α_1 could be swapped, as they filter completely distinct cases. The body of clause α_0 is easy to guess: it must be `Q`, since it corresponds to the case when we want to append `Q` to the empty list:

$$\begin{array}{l}
\texttt{join(\ \ \ [],Q,\ \ \ [])} \xrightarrow{\alpha_0} \mathbf{\texttt{Q}}\texttt{;}\\
\texttt{join(\ \ \ [],Q,[I|A])} \xrightarrow{\alpha_1} \boxed{}\texttt{;}\\
\texttt{join([I|P],Q,\ \ \ \ A)} \xrightarrow{\beta} \texttt{join(P,Q,[I|A]).}
\end{array}$$

How do we relate `Q`, `I` and `A` in clause α_1 with the result we are looking for? Given the call `join(P,Q,[])`, we know that `[I|A]` is `P` reversed, so item `I` is last in `P` and it should be on top of `Q` in the result. Now, we found above that `[I|Q]` is the end of the result. But what should we do with `A`? The key is to realise that we need to start the same process again, that is, we need another recursive call:

```
join(   [],Q,   []) $\xrightarrow{\alpha_0}$ Q;
join(   [],Q,[I|A]) $\xrightarrow{\alpha_1}$ join([],[I|Q],A);
join([I|P],Q,    A) $\xrightarrow{\beta}$ join(P,Q,[I|A]).
```

To test the correctness of this definition, we can try a small example:

```
join([1,2,3],[4,5],[]) $\xrightarrow{\beta}$ join([2,3],      [4,5],    [1])
                       $\xrightarrow{\beta}$ join(  [3],      [4,5],  [2,1])
                       $\xrightarrow{\beta}$ join(   [],      [4,5],[3,2,1])
                       $\xrightarrow{\alpha_1}$ join(   [],    [3,4,5],  [2,1])
                       $\xrightarrow{\alpha_1}$ join(   [],  [2,3,4,5],    [1])
                       $\xrightarrow{\alpha_1}$ join(   [],[1,2,3,4,5],     [])
                       $\xrightarrow{\alpha_0}$ [1,2,3,4,5].
```

As a conclusion, the tail form version of `join/2`, called `join_tf/2`, requires an auxiliary function `join/3` with an accumulator whose purpose is to reverse the first argument:

```
join_tf(P,Q)        $\xrightarrow{\alpha}$ join(P,Q,[]).
join(   [],Q,   []) $\xrightarrow{\beta}$ Q;
join(   [],Q,[I|A]) $\xrightarrow{\gamma}$ join([],[I|Q],A);
join([I|P],Q,    A) $\xrightarrow{\delta}$ join(P,Q,[I|A]).
```

We also know what to do when the control context is not made of a call to some associative operator: push the variables it contains and when the input list is empty, pop them and apply the context.

Efficiency. The number of steps to rewrite `join_tf(`P`,`Q`)` into a value is greater than with `join(`P`,`Q`)`, as we guessed while writing the previous example. Indeed, assuming that `P` contains n items, we have

- one step to obtain `join(`P`,`Q`,[])`, by clause α;
- n steps to reverse P in the accumulator, by clause δ;
- n steps to reverse the accumulator on top of Q, by clause γ;
- one step when the accumulator is finally empty, by clause β.

Thus, the total number of steps is $2n + 2$, which is twice the delay of the previous version. When we wrote two different versions of the factorial function, `fact_tf/1` in tail form and `fact/1`, the number of rewrites was, respectively, $n+1$ versus n. So why is the difference much greater between `join/2` and `join_tf/2`? The explanation is found in the heterogeneous natures of the accumulators. In the case of the factorial, the operation applied to the accumulator is a multiplication and the accumulator is, at all times, a partial result. In the case of appending a list to another, the operation applied to the accumulator consists in pushing an item onto a list and has to be undone later: the accumulator is not a partial result but a *temporary list* used to hold the items of the first list in reverse order. We shall find many occurrences of this situation. Meanwhile, it is important to remember that a tail form variant of a function operating on lists may lead to a slower program, even though the call stack remains of constant size. Also, the derived definition in tail form may be longer, as illustrated by `join_tf/2`: four clauses instead of two.

It may be argued that any instance of a control context has to be stored in the call stack and later removed to be combined with the result of said call, therefore, since we only measure the number of user-defined rewrites, these removals from the call stack are not accounted for, despite incurring a delay. This is a valid point, but this delay is arguably much shorter than the allocation time because, for reasons we will expand in a later chapter, deallocation of data in the heap takes significantly more time than in the call stack.

As a final note on efficiency, we can summarise the *execution trace* above as a product (composition) of clauses: $\alpha \cdot \delta^n \cdot \gamma^n \cdot \beta$, or, simply, $\alpha\delta^n\gamma^n\beta$. The number of clauses $\mathcal{D}_n^{\texttt{sum\_tf}}$ of `sum_tf(`L`)`, when the list L contains n integers, is then the length of the trace, given that the length of a clause c, noted $|c|$, is 1:

$$\begin{aligned}\mathcal{D}_n^{\texttt{sum\_tf}} &:= |\alpha\delta^n\gamma^n\beta| = |\alpha| + |\delta^n| + |\gamma^n| + |\beta| = 1 + |\delta| \cdot n + |\gamma| \cdot n + 1\\ &= 2n + 2.\end{aligned}$$

Polymorphism. The function `join/2` presents a characteristic which makes it different from `fact/1`, `sum/1` and `mult/1` because the nature, usually called *type*, of the items in the lists to be joined is irrelevant. For example, we showed previously that `join([3,[]],[[]])` can be computed, in spite the lists to be joined containing integers and/or lists. In the case of the other mentioned functions, the argument must either be an integer or a list containing integers, otherwise the computation fails because the involved arithmetic operators can not be applied to something else other than an integer. When the evaluation of a function call never depends on the type of (some part of) its arguments, the function is said to be *polymorphic*. This does not mean that any type would fit a polymorphic function: `join/2` does require its first argument to be a list, what does not matter is the type of the items it contains.

Function definitions as metadata. The previous discussions on obtaining equivalent definitions which are in tail form suppose to consider programs as data. At this point, it is a methodological standpoint only and we do not mean that functions can be processed as lists (we shall come back on this later), but, more informally, we mean that definitions can be transformed into other definitions and that this is often an excellent method, as opposed to trying to figure out from scratch the final definition. It would have been probably more difficult to write the tail form variant of `join/2` without having first designed the version not in tail form. It is in general not a good idea to start head-on by defining a function in tail form because it may either be unnecessary, the maximum allowed size for the call stack being large enough, or lead to a mistake since these kinds of definitions are usually more involved.

Sometimes the result is correct but overly involved. Let us consider a simple case by defining a function `last/1` such that `last(`L`)` computes the last item of the non-empty list L. The correct approach is to forget about tail forms and aim straight at the heart of the problem. We know that L can not be empty, so let us start with the following head:

```
last([I|L]) -> □.
```

Can we reach the result in one step? No, because we do not know whether `I` is the sought item: we need to know more about `L`. This additional information about the structure of `L` is given by more precise heads: `L` can be empty or not. This is

```
last([I|   []]) -> □;
last([I|[J|L]]) -> □.
```

The first head can be simplified as follows:

```
last(      [I]) -> □;
last([I|[J|L]]) -> □.
```

The first body is easy to guess:

```
last(      [I]) -> I;
last([I|[J|L]]) -> □.
```

In the last clause, how do `I`, `J` and `L` relate to the result? Can we reach it in one step? No, despite we know that `I` is *not* the result, we still don't know whether `J` is, so we have to start over again, which means a recursive call is required:

```
last(      [I]) -> I;
last([I|[J|L]]) -> last(□).                 % I is useless
```

Note how knowing that some specific part of the input is not useful to build the output is useful knowledge. By the way, Erlang allows us to silence these useless parts by writing instead an underscore as in

```
last(      [I]) -> I;
last([_|[J|L]]) -> last(□).           % Anonymous list head
```

We can not call recursively `last(L)` because `L` may be empty and the call fail, which would mean that the answer was `J`. Therefore, we must call with `[J|L]` to give `J` the opportunity to be the last:

```
last(      [I]) -> I;
last([_|[J|L]]) -> last([J|L]).
```

As we advocated previously, the next phase consists in testing this definition for correctness and completeness, using meaningful examples. Let us assume that we are convinced that this definition is correct and complete. What's next? Let us try to improve upon it. Let us look for common patterns that can be simplified. For instance, we observe that `[J|L]` is used as a whole, in other words, `J` and `L` are not used separately in the second body. Therefore, it is worth trying to replace the pattern by a more general one, in this case, a variable:

```
last(  [I]) -> I;
last([_|L]) -> last(L).                          % L is non-empty
```

This transformation is correct because the case where `L` is empty has already been matched by the first head. Notice also that we considered a definition as some data (We should write more accurately *metadata* since definitions are not data processed by the program, but by the programmer.) which we transformed, step by step, making sure that each step starts at a correct and complete definition and leads to an equivalent one, so a series of transformations compulsorily yields an equivalent function. In passing, it is now obvious that the definition we found for `last/1` is in tail form.

What if we had tried to find directly a definition in tail form? We might have recalled that such definitions need an accumulator and we would have tried perhaps something along these lines:

```
last(L)          -> last__(L,0).
last__(   [],A) -> A;
last__([I|L],A) -> last__(L,I).
```

The first observation may be about the function name `last__`. Why not write the following, in accordance with the style up to now?

```
last(L)        -> last(L,0).
last(   [],A) -> A;
last([I|L],A) -> last(L,I).
```

There would be no confusion between `last/1` and `last/2` because, each taking a different number of arguments, they are logically considered different in Erlang. The reason why we recommend to distinguish the names and, in general, to use one name for only one function (inside a module), is that this discipline enables the compiler to catch the error consisting in forgetting one argument. For instance, the program

```
last(L)       -> last(L,0).
last(   [],A) -> A;
last([I|L],A) -> last(L).           % Argument silently missing
```

contains an error that goes unreported, while

```
last(L)         -> last__(L,0).
last__(   [],A) -> A;
last__([I|L],A) -> last__(L).                     % Error reported
```

raises an error in the compiler, which allows us to realise an error was made and helps in fixing it. However, for didactic purposes in this book, we will not always follow this recommendation of having unique function names. The possibility to use the same name for different functions which can be otherwise distinguished by the number of their arguments is called *overloading*. Overloading of methods in the programming languages Java and C++ is allowed, but the rules used to distinguish amongst the different methods sharing the same name is different than in Erlang, as it makes use of the number of parameters but also their static types.

Computing the call `last([1,2,3])` with the original definition, we find that the three clauses are covered until the correct result is found, that is, `3`. Because we recommended previously to make some litmus test and the argument is a list, we try the empty list and find that `last([])` $\rightarrow$ `last__([],0)` $\rightarrow$ `0`, which is unexpected, since this test ought to fail (`last/1` is not defined for the empty list). Can we fix this case? Yes, this can be done in a simple way: let us change the head of `last/1` so that only non-empty lists are matched. We find here a case where more information on the structure of the input is needed and a variable is too general a pattern. We need instead

```
last([I|L])     -> last__([I|L],0).                         % Fixed
last__(   [],A) -> A;
last__([I|L],A) -> last__(L,I).
```

This emendation seems to go against an improvement we made earlier, when we replaced `[J|L]` by `L`, but it does not: here we want to exclude some input, that is, we do not seek an equivalent function, whilst before the purpose was to simplify and obtain an equivalent function.

This last definition of `last/1` is correct and complete but a careful review should raise some doubts about its actual simplicity. For example, the initial value of the accumulator, given in the unique body of `last/1` is `0`, but this value is never used, because it is discarded immediately after in the second body of `last__/2`, as it is conspicuous if we write instead the equivalent

```
last([I|L])     -> last__([I|L],0).                 % Why zero?
last__(   [],A) -> A;
last__([I|L],_) -> last__(L,I).                 % A was useless
```

Indeed, we could write the equivalent definition:

```
last([I|L])     -> last__([I|L],7).                 % Why not 7?
last__(   [],A) -> A;
last__([I|L],_) -> last__(L,I).
```

The initial value of the accumulator here does not even need to be an integer, it could be of whatever type, like `[4,[]]`. We should better give up this overly complicated definition, which is the result of a methodology that does not consider programs as data and is founded on the wrong assumption that definitions in tail form always require an accumulator: they do not, in general. Take for example the polymorphic identity: `id(X) -> X`. It is trivially in tail form. In passing, being in tail form has nothing to do, in general, with recursion, despite the widespread and unfortunate locution “tail-recursive function.” A recursive definition may be in tail form, but a definition in tail form may not be recursive, as `id/1`.

List reversal. How do we reverse a given list? Let us explore two approaches: the first one tries to analyse the problem in terms of a previous problem, the second reuses, more abstractly, some insight gained in solving a previous larger or similar problem. Let `srev/1` be the function such that `srev(`L`)` is rewritten in a list which contains the items of list L in reverse order. It is easy to write the heads:

```
srev(   []) -> ▭;
srev([I|L]) -> ▭.
```

We must ponder whether `srev/1` is defined for the empty list, since the original question left that point out. It was said that the result is a list containing all the items of the input and, since the input contains no item, it implies that the result is a list that contains no items. This makes sense as this is the very definition of the empty list:

```
srev(   []) -> [];
srev([I|L]) -> ▭.
```

We must wonder now if we can reach the result in one step in the remaining body. Clearly no, because we have to reverse `L`, that is, a recursive call `srev(L)` must be placed somewhere. Where is the item `I` located in the final result? Since the purpose is to reverse `[I|L]`, we deduce that `I` must be located at the end of the resulting list. So the situation so far is as follows:

```
srev(   []) -> [];
srev([I|L]) -> □ srev(L) □ I □.
```

As usual, the empty boxes are a reminder of some relationship to be found between the pieces already in place. For instance, we know that the body must be a list, so should we try the following?

```
srev(   []) -> [];
srev([I|L]) -> [srev(L) | I].                          % Hmm...
```

For this to work we need to check the types of the expressions involved at this point. We know that `srev(L)` is a list made of items from `L`, because it is a recursive call; item `I` is an item from `[I|L]`. But the position of `srev(L)` in our proposal is that of the head of a list and `I` is the tail of the same, which is wrong: they should be reversed to be homogeneous. But the relative order of `srev(L)` and `I` above is correct, so we must think of some other relationship. What we want is to join `srev(L)` and `I`, which reminds us of the function `join/2`:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),I).                      % Still...
```

Again, let us check the types. We know that in `join(`P`,`Q`)`, both P and Q must be lists. But `I` is an item, so it does not fit in as it is. The solution consists in making a singleton from `I`:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

Before we test this definition, a keen observer may have raised an objection to the last argument: it is possible that `I` be a list. Indeed, but what was exactly meant was that `I` is considered as an item *with respect to `srev(L)`*, not in an absolute sense. This is best understood by looking at the abstract syntax tree of the input list. Let us take for instance `[3,[1,[]],[]]`. This value is a short-hand for `[3|[[1|[[]|[]]]|[[]|[]]]]`, whose tree is given in FIGURE 11 on the following page. Following the rightmost edges from the root to the rightmost `[]` (the dashed line), the left subtrees (the solid lines) are the items of the embedding list `[3,[1,[]],[]]`: `3`, `[1,[]]` and `[]`. It does not matter that one item is an integer while the other two are lists, what does matter is their relative positions in the abstract syntax tree because it defines their interpretation as items of the whole list. Perhaps we can also remark that the definition of `srev/1` we came upon contains a call inside a call. We found on page 16 a similar situation

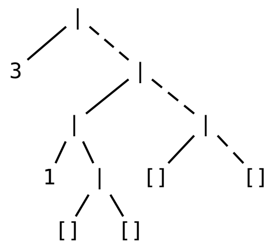


FIGURE 11: Abstract Syntax Tree of `[3,[1,[]],[]]`

with a *naïve* definition of the Fibonacci function:

```
fib(0)            -> 1;
fib(1)            -> 1;
fib(N) when N > 1 -> fib(N-1) + fib(N-2).
```

Remember that (`+`) is an operator, that is, a predefined function, so its arguments are function calls themselves: `fib(N-1)` and `fib(N-2)`. When multiple calls are present in a body, there could be several ways to rewrite it, but we saw that the strategy used by Erlang consists in rewriting first the most embedded calls and then their immediately embedding calls etc. until the outermost call is evaluated. But the order in which two calls which are arguments to the same function or arithmetic operator are evaluated is not specified in Erlang. In the case of `srev/1`, there is no choice and the first call to be computed is `srev(`L`)`, because it is the most deeply embedded, and then the immediately embedding `join(`$\overline{L}$`,[`I`])`, where $\overline{L}$ stands for the value of `srev(`L`)` previously computed. In the case of `fib/1`, it is purposefully left unspecified which call of `fib(N-1)` or `fib(N-2)` is rewritten first.

Another observation is that our definition of `srev/1` is not in tail form: there is a control context `join(␣,[`I`])` around the recursive call `srev(`L`)`. (Let us keep in mind that the fact that the call is recursive does not matter here.) Therefore, we should expect the maximum size of the call stack to grow as the input size grows. Let us take as an example `srev([3,2,1])`, whose expected outcome must be `[1,2,3]`, and let us follow the rewrites on the abstract syntax trees by drawing the node to be rewritten at each step in a box. Let us label the clauses defining `srev/1` with the Greek letters α and β:

$$\texttt{srev(\ \ \ [])} \xrightarrow{\alpha} \texttt{[];}$$
$$\texttt{srev([I|L])} \xrightarrow{\beta} \texttt{join(srev(L),[I]).}$$

FIGURE 12 on the facing page shows the first rewrites until there are no more calls to `srev/1`. Let us label the clauses defining `join/2` as follows

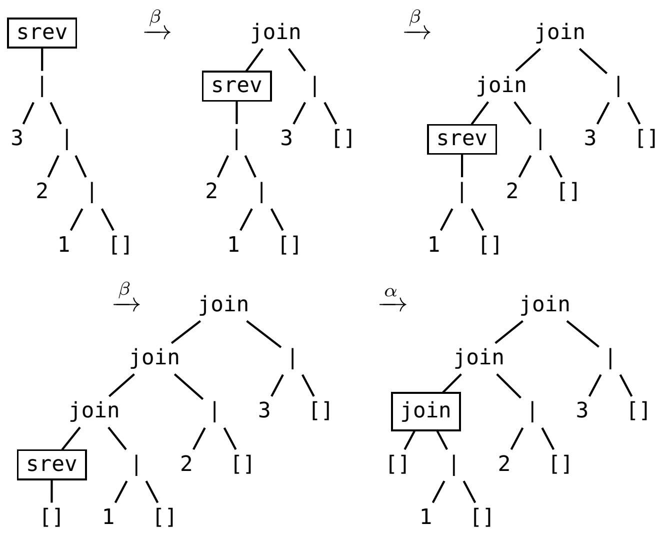


FIGURE 12: $\texttt{srev([3,2,1])} \twoheadrightarrow \texttt{join(join(join([],[1]),[2]),[3])}$

$$\begin{aligned}
&\texttt{join(\ \ \ [],Q)} \xrightarrow{\gamma} \texttt{Q;}\\
&\texttt{join([I|P],Q)} \xrightarrow{\delta} \texttt{[I|join(P,Q)].}
\end{aligned}$$

and resume the rewrites in FIGURE 13 on the next page. This computation can also be followed in a flat representation as well:

$$\begin{aligned}
\texttt{srev([3,2,1])} &\xrightarrow{\beta} \texttt{join(srev([2,1]),[3])}\\
&\xrightarrow{\beta} \texttt{join(join(srev([1]),[2]),[3])}\\
&\xrightarrow{\beta} \texttt{join(join(join(srev([]),[1]),[2]),[3])}\\
&\xrightarrow{\alpha} \texttt{join(join(join([],[1]),[2]),[3])}\\
&\xrightarrow{\gamma} \texttt{join(join([1],[2]),[3])}\\
&\xrightarrow{\delta} \texttt{join([1|join([],[2])],[3])}\\
&\xrightarrow{\gamma} \texttt{join([1|[2]],[3])}\\
&= \texttt{join([1,2],[3])}\\
&\xrightarrow{\delta} \texttt{[1|join([2],[3])]}\\
&\xrightarrow{\delta} \texttt{[1|[2|join([],[3])]]}\\
&= \texttt{[1,2|join([],[3])]}\\
&\xrightarrow{\gamma} \texttt{[1,2|[3]]}\\
&= \texttt{[1,2,3].}
\end{aligned}$$

There are 10 steps: 4 with clauses α and β (`srev/1`), plus 6 with the clauses γ and δ (`join/2`). The contents of the call stack, at any time, is the branch from the root to the boxed node, which is the call to be rewritten next, as shown in FIGURE 14 on page 57 (the final downward arrow is a reference to the result in the heap). We see that the longest extant of the call stack was reached after the third step, when $\boxed{\texttt{srev}}$ is the deeper in the abstract syntax tree. As we shall understand, the occurrence of the number 3 here is no mere coincidence.

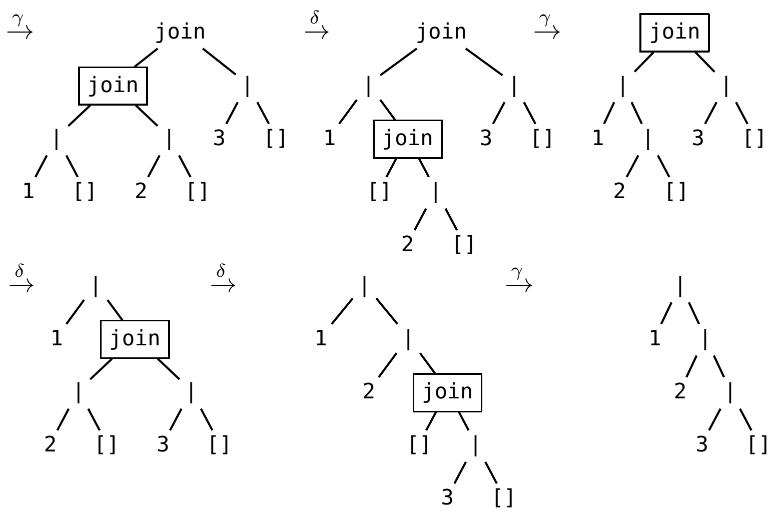


FIGURE 13: `join(join([1],[2]),[3])` $\twoheadrightarrow$ `[1,2,3]`

What would have happened, had we chosen `join_tf/2` (tail form) instead of `join/2`? Let us recall the definition of `join_tf/2`:

```
join_tf(P,Q)         -α-> join(P,Q,[]).
join(   [],Q,   []) -β-> Q;
join(   [],Q,[I|A]) -γ-> join([],[I|Q],A);
join([I|P],Q,    A) -δ-> join(P,Q,[I|A]).
```

and define an alternative definition `srev_alt/1` as follows:

```
srev_alt(   []) -ε-> [];
srev_alt([I|L]) -ζ-> join_tf(srev_alt(L),[I]).
```

Then, our example unfolds as follows.

```
srev_alt([3,2,1])
   -ζ-> join_tf(srev_alt([2,1]),[3])
   -ζ-> join_tf(join_tf(srev_alt([1]),[2]),[3])
   -ζ-> join_tf(join_tf(join_tf(srev_alt([]),[1]),[2]),[3])
   -ε-> join_tf(join_tf(join_tf([],[1]),[2]),[3])
   -α-> join_tf(join_tf(join([],[1],[]),[2]),[3])
   -β-> join_tf(join_tf([1],[2]),[3])
```

$\xrightarrow{\alpha}$ `join_tf(join([1],[2],[]),[3])`
$\xrightarrow{\delta}$ `join_tf(join([],[2],[1]),[3])`
$\xrightarrow{\gamma}$ `join_tf(join([],[1,2],[]),[3])`
$\xrightarrow{\beta}$ `join_tf([1,2],[3])`
$\xrightarrow{\alpha}$ `join([1,2],[3],[])`
$\xrightarrow{\theta}$ `join([2],[3],[1])`
$\xrightarrow{\delta}$ `join([],[3],[2,1])`
$\xrightarrow{\delta}$ `join([],[2,3],[1])`
$\xrightarrow{\gamma}$ `join([],[1,2,3],[])`
$\xrightarrow{\beta}$ `[1,2,3]`.

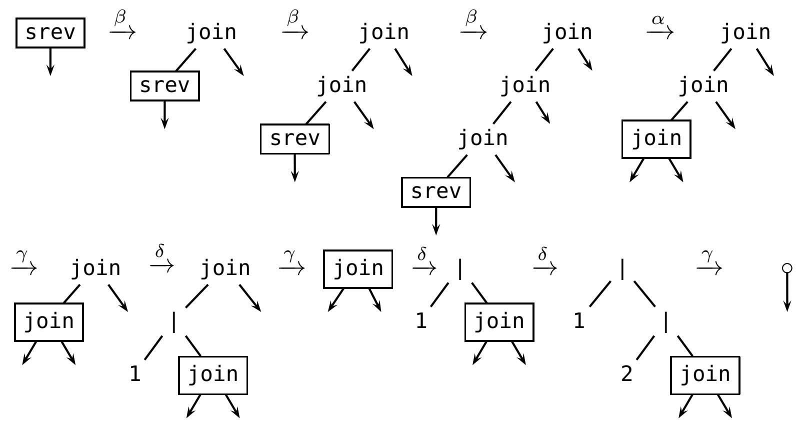


FIGURE 14: Call stack of FIGURES 12 to 13 on pages 55–56

These rewrites can also be viewed two-dimensionally as abstract syntax trees in FIGURE 15 on page 58, where the last step ($\xrightarrow{\epsilon}$) of `srev_alt/1` is shown first. There are 16 steps: 4 with the definition of `srev_alt/1`, plus 12 with the definition of `join/2`. The contents of the call stack, at any time, is the branch from the root to the boxed node in FIGURE 16 on page 59 (steps 1–4 are omitted as they are already displayed in FIGURE 14).

What can we conclude from using `join/2` to define `srev/1` and `join_-tf/1` to define `srev_alt/1`?

On the example `srev_alt([3,2,1])`, using `join_tf/2` (tail form) leads to longer rewrites (16 against 10) and the maximum call stack capacity is the same (since it is reached after the same third step in both cases). We may not be surprised to find a greater delay by using a tail form definition, but it may come as a surprise that there is no gain in the call stack allocation.

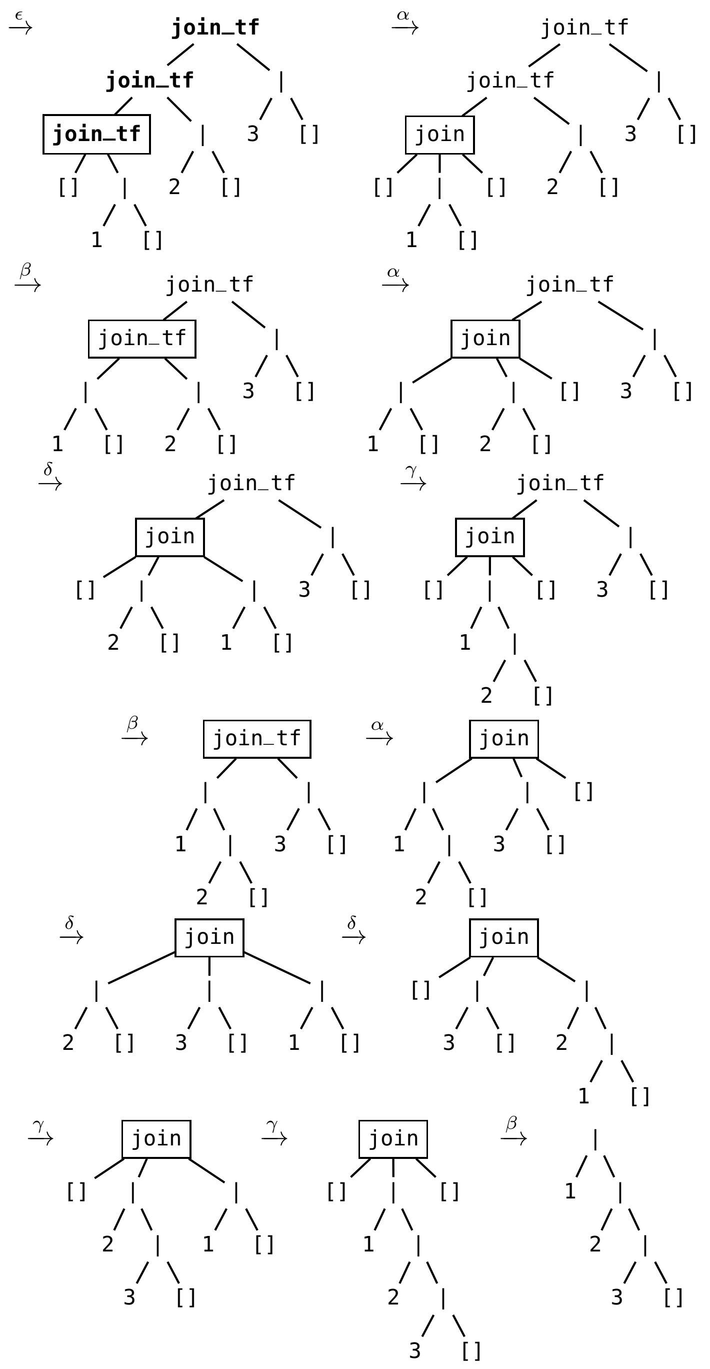


FIGURE 15: join_tf(join_tf(join_tf([],[1]),[2]),[3])↠[1,2,3]

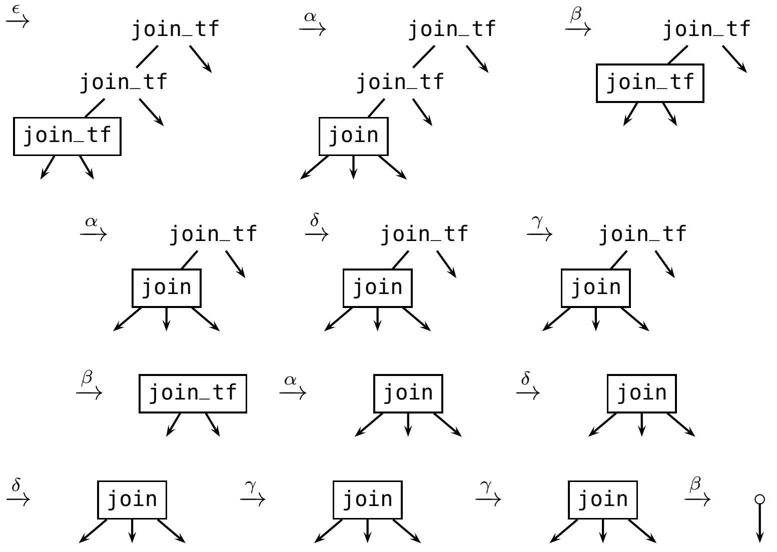


FIGURE 16: Call stack of FIGURE 15 on the next page

Is perhaps the reason that the test was carried out on a list too short to show otherwise? When considering such a question, instead of trying a longer input by hand, it is worth revisiting the previous test and try to generalise the process for a list of n items: `[`I_0`,...,`I_{n-1}`]`. The third rewrite, involving only the definition of `srev/1`, leads to the abstract syntax tree in FIGURE 17 on page 60, where the dotted edge stands for the repetition of the same schema: a node `join` with a right subtree `[`I_i`]`. The number of nodes in the call stack is thus $n + 1$ and this holds whichever version for joining two lists, `join/2` or `join_tf/2`, is chosen. Therefore, a lesson to remember from this short analysis is that the usage of a definition in tail form does not automatically lead to gains in terms of call stack allocation or speed—indeed, a slowdown is likely. One way to recover the benefit of definitions in tail form is to make sure that *all* the functions that are composed are either in tail form or do not use call stack space at least in proportion of the input size.

Therefore, let us turn our attention back to the design of `srev/1`. We would like to find a definition equivalent to `srev/1` but in tail form. There are at least two options left open: either start from scratch or try to modify the program which is correct and complete but not in tail form. Another way to analyse the problem consists in realising that

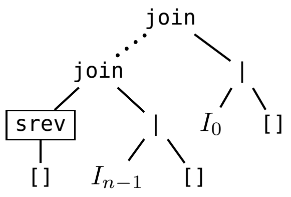


FIGURE 17: `srev([`I_0`,...,`I_{n-1}`])` after n rewrites

the definition of `join_tf/2` already includes the concept of list reversal. In other words, instead of calling `join/2` in the definition of `srev/1`, let us realise that the design of `srev/1` is already embedded in the definition `join_tf/2` and write accordingly another version of `srev/1`, called `rev/1` in tail form. It is important to realise that `rev/1` is not derived mechanically from `srev/1`: it implements another design. The definition on page 47 shows that clause δ has the effect of reversing the first argument into the accumulator (the third argument). Moreover, clause γ reverses the accumulator on the second argument. Thus, we already know how to reverse a list: simply use an extra argument to accumulate the items of the list to be reversed, since the accumulation consists in pushing the items one by one on top of the initially empty list, the resulting value of the accumulator is the input list reversed. In Erlang, this is implemented as follows:

```
rev(L)             -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

All the clauses are in tail form, hence is the whole definition. The accumulator is named `Q`. We must bear in mind that the reason why this definition is efficient is not due to it being in tail form, but because the algorithm, that is, the underlying idea, is better.

Exercises. [Answers on page 369.]

1. Define a function `len_tf/1` in tail form such that `len_tf(`L`)` is rewritten to the number of items in the list L. Do you need an auxiliary function? How many rewrites are needed to reach the result when L contains n items?
2. Define a function `penul/1` such that `penul(`L`)` is rewritten to the penultimate item in list L. Make sure to avoid any confusion in the syntax of lists between the comma (`,`) and the vertical bar (`|`). Test for correctness and completeness. If L contains n items, how

many rewrites are needed to reach the result? Is your definition in tail form?

3. Define a function `rep_fst/1` such that the call `rep_fst(`L`)`, where L is a list, is rewritten into a list of same length as L but containing only the first item of L repeated.
4. Same with `rep_lst/1`, but repeating the last item instead of the first.
5. Consider the following alternative definition for reversing a list.

```
rev_bis(   []) -> [];
rev_bis([I|L]) -> join(rev_bis(L),[I],[]).

join(   [],Q,   []) -> Q;
join(   [],Q,[I|A]) -> join([],[I|Q],A);
join([I|P],Q,    A) -> join(P,Q,[I|A]).
```

Lay out the rewrites of `rev_bis([3,2,1])` with this definition, both with the abstract syntax trees and the flat representation. Compare with the definition of `srev/1`:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).

join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

Chapter 3

Delay

When a function call is computed, several outcomes are possible: the rewrites may

- never end (*loop*),
- end with a memory (call stack or heap) overflow,
- end with a type error (as `1 + []`),
- end with a match failure (a body contains a call which is matched by no head),
- end with a *value* (that is, an expression which is a constant, for example, a list or an integer, and therefore cannot be further rewritten).

In the latter case, it is sometimes feasible to express the number of rewrites needed to reach the result in terms of the input size. Accordingly, a loop requires an infinite number of steps and a value none. Also, a measure of the input must be decided upon in order to be able to apprehend the number of rewrite steps, which, in turn, can be interpreted temporally as a *delay* (between the moment when the computation starts and when it ends successfully with a value). The choice of a measure of the data depends entirely on the programmer, but it is generally considered meaningful, for instance, to take the number of items in a list as its size, and an Erlang integer `N` is measured by its mathematical correspondent $n \in \mathbb{N}$. The length of a list, that is, the number of items it contains, can be defined in several ways in Erlang, one being

```
len(   []) -> 0;
len([_|L]) -> 1 + len(L).
```

It does not matter here that this definition is not in tail form because its purpose is to prove how a peculiar measure on lists can easily be implemented.

Given an expression e, we denote the number of rewrites to reach its value, assuming it exists, by $[\![e]\!]$ and the value itself by $\overline{e}$. The strategy of Erlang, which consists in computing the arguments of a call before the call itself and then selecting clauses for matching according to the order of definition, implies that if a value exists, then it is unique. Hence, computing a value $\overline{v}$ incurs a null delay: $[\![\overline{v}]\!] = 0$. Moreover, the delay of a function call is the sum of the delays of its arguments plus the delay of rewriting the call with the resulting values. For example, let P and Q be two expressions whose values are lists. Then the delay of $\texttt{join(}P\texttt{,}Q\texttt{)}$ is denoted by $[\![\texttt{join(}P\texttt{,}Q\texttt{)}]\!]$ and equals the sum of the delays of P and Q, plus the delay of $\texttt{join(}\overline{P}\texttt{,}\overline{Q}\texttt{)}$:

$$[\![\texttt{join(}P\texttt{,}Q\texttt{)}]\!] := [\![P]\!] + [\![Q]\!] + [\![\texttt{join(}\overline{P}\texttt{,}\overline{Q}\texttt{)}]\!].$$

We need now to connect these notions to function definitions: the delay of a call matching a head is 1 plus the delay of the corresponding body in the clause.

Joining two Lists. Let us consider the definition of `join/2` which is not in tail form:

```
join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

Applying the previous definition of the delay to each clause leads straightforwardly to the following *recurrence equations*. Roughly speaking, "recurrence" is the mathematical alter ego of "recursive" in computer science.

$$\begin{aligned}[\![\texttt{join([],Q)}]\!] &:= 1 + [\![\texttt{Q}]\!],\\ [\![\texttt{join([I|P],Q)}]\!] &:= 1 + [\![\texttt{[I|join(P,Q)]}]\!].\end{aligned}$$

The relation $a := b$ means "a equals b by definition;" $a = b$ means "a equals b by means of some mathematical deduction" and $a \equiv b$ means "Erlang expressions a and b are rewritten into the same value," more precisely, their abstract syntax trees are identical. Here, all Erlang variables, `I`, `P` and `Q`, denote values, since they are bound to the arguments during the matching of the call. But what about $[\![\texttt{[I|join(P,Q)]}]\!]$? The Erlang operator (`|`) can be considered as a function which is not defined by any clause, therefore it incurs no delay and only the delays of its two arguments matter. A more detailed analysis would take into account the time needed to push an item on top of a list, but we assume here that

$$[\![\texttt{[I|L]}]\!] = [\![\texttt{I}]\!] + [\![\texttt{L}]\!].$$

Therefore

$$[\![\texttt{[I|join(P,Q)]}]\!] = [\![\texttt{I}]\!] + [\![\texttt{join(P,Q)}]\!].$$

We have $[\![\texttt{I}]\!] = 0$ and the equations are equivalent to

$$[\![\texttt{join([],Q)}]\!] = 1, \quad [\![\texttt{join([I|P],Q)}]\!] = 1 + [\![\texttt{join(P,Q)}]\!].$$

Note that we must use here $(=)$ instead of $(:=)$. We need now to define the delay in terms of the size of the input, that is, the arguments. The function call $\texttt{join(}P\texttt{,}Q\texttt{)}$ has two lists as arguments. The usual way to measure the size of a list is to consider the number of elements it contains, that is, its length. Here, the size of the input is characterised by a pair (n, m), where $n, m \in \mathbb{N}$ are the mathematical abstractions of the values of $\texttt{len(}P\texttt{)}$ and $\texttt{len(}Q\texttt{)}$. For practical purposes, we mean to identify Erlang integers and mathematical integers. Let us define the delay of `join/2` in terms of the input size as

$$\mathcal{D}^{\texttt{join}}_{n,m} := [\![\texttt{join(}P\texttt{,}Q\texttt{)}]\!],$$

where n is the length of P and m is the length of Q. Using this definition, the previous equations are equivalent to

$$\mathcal{D}^{\texttt{join}}_{0,q} = 1, \quad \mathcal{D}^{\texttt{join}}_{p+1,q} = 1 + \mathcal{D}^{\texttt{join}}_{p,q}.$$

where `P` and `Q` are supposed to contain respectively p and q items. It is not too late to observe or recall that `Q` is constant all along the rewrites, which can be seen here too because all the occurrences of q are q itself. In other words: $q = m$. Thus, for the sake of clarity, let us define $\mathcal{D}^{\texttt{join}}_{p} := \mathcal{D}^{\texttt{join}}_{p,m}$; then we simply have

$$\mathcal{D}^{\texttt{join}}_{0} = 1, \quad \mathcal{D}^{\texttt{join}}_{p+1} = 1 + \mathcal{D}^{\texttt{join}}_{p}.$$

As we wish to express $\mathcal{D}^{\texttt{join}}_{n}$ in terms of n alone, that is, we want a *closed form*, let us write the first terms and guess the general form when $p + 1 = n$ in the second equation:

$$\begin{aligned}
\mathcal{D}^{\texttt{join}}_{0} &= 1,\\
\mathcal{D}^{\texttt{join}}_{1} &= 1 + \mathcal{D}^{\texttt{join}}_{0} = 1 + 1 = 2,\\
\mathcal{D}^{\texttt{join}}_{2} &= 1 + \mathcal{D}^{\texttt{join}}_{1} = 1 + 2 = 3,\\
&\vdots\\
\mathcal{D}^{\texttt{join}}_{n} &= 1 + \mathcal{D}^{\texttt{join}}_{n-1} = 1 + n.
\end{aligned}$$

But we must check if our guess is correct by replacing $\mathcal{D}^{\texttt{join}}_{p} = 1 + p$ back into the original equations, which must be satisfied:

$$\mathcal{D}^{\texttt{join}}_{0} = 1 + 0 = 1, \quad \mathcal{D}^{\texttt{join}}_{p+1} = 1 + (p + 1) = 1 + \mathcal{D}^{\texttt{join}}_{p}.$$

And they are. So we can trust that $\mathcal{D}^{\texttt{join}}_{n} = n + 1$.

We would remiss if we do not mention that, whilst the previous reckoning is perfectly valid, there exists a direct approach that does not suppose to guess the result and later check it. The technique needed here, called *telescoping*, consists in forming the differences between two successive terms in the series $(\mathcal{D}_p^{\texttt{join}})_{p\in\mathbb{N}}$ and then summing these differences. The second equation implies the following ones:

$$
\begin{aligned}
\mathcal{D}_{p+1}^{\texttt{join}} - \mathcal{D}_p^{\texttt{join}} &= 1, && \text{where } p \geqslant 0,\\
\sum_{p=0}^{n-1}\left(\mathcal{D}_{p+1}^{\texttt{join}} - \mathcal{D}_p^{\texttt{join}}\right) &= \sum_{p=0}^{n-1} 1, && \text{where } n > 0,\\
\mathcal{D}_n^{\texttt{join}} - \mathcal{D}_0^{\texttt{join}} &= n,\\
\mathcal{D}_n^{\texttt{join}} &= n + 1. && (3.2)
\end{aligned}
$$

The clue that the reasoning has been tidied up is that it contains no more ellipses (...), English or Latin locutions like "and so on" and "etc." Furthermore, it is important to have a clear understanding of the behaviour of the delay when the input becomes significantly large with res<pect to the chosen measure. This insight is provided by considering an *equivalent function* when the variable takes arbitrarily large values:

$$\mathcal{D}_n^{\texttt{join}} \sim n, \text{ as } n \to \infty.$$

By definition, two functions $f(n)$ and $g(n)$ are *asymptotically equivalent* if $\lim_{n\to\infty} f(n)/g(n) = 1$. Our assumption all along has been that the number of steps necessary to fully rewrite a function call, which we call delay and other authors often call *time complexity* or *cost*, is proportional to the real, wall-clock, delay. This means, for example, that the time required to join two lists is almost proportional to the length of the first list and the more the list grows, the more accurate the proportionality becomes. In particular, doubling the size of the first list will approximately double the time to get the result, whereas doubling the length of the second list will have no impact at all. Despite being quite reasonable, this proportionality assumption is slightly weak because many untold details have been abstracted away so as to get a simple introduction to functional programming. A much stronger case is found in comparing two functions, especially if they compute the same results on the same inputs (this is another sort of equivalence). Indeed, both functions then assume the same abstract execution model (think of the implementation of some virtual machine, for example) and no direct reference to the physical world is needed for validating the comparison: knowing which function is faster *in the same environment* implies that this function is also faster with respect to the wall-clock. This kind of question can sometimes be answered very reliably and the asymptotic expressions, that is, involving the equivalence sign ($\sim$), are especially useful because real life data is usually big, assuming that the measure chosen is pertinent. If small inputs are likely, it can be wise to have a version of the function dedicated to be efficient on such data and another version specialised for large cases.

Is it mandatory to go through all these mathematics to reach the result? Most of the time, such thorough investigation is not necessary and sometimes not even tractable. We could have think as follows. Let

```
join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

In the calls `join(`P`,`Q`)`, we know that the size of Q does not matter. By focusing on the recursive call in the definition, we see that the first parameter changes from `[I|P]` (in the head) to `P`. Thus, the sensible measure on lists is the length because it decreases by one at each recursive call. Let us suppose that `P` is bound to an argument which contains p items. Let $\mathcal{D}_p^{\texttt{join}}$ be the number of steps to rewrite `join(`P`,`Q`)` to a value. Then the second clause implies that $\mathcal{D}_{p+1}^{\texttt{join}} = 1 + \mathcal{D}_p^{\texttt{join}}$ and the first one $\mathcal{D}_0^{\texttt{join}} = 1$. The number 1 in each case comes from the fact that one clause is used (an arrow "`->`"). Then the recurrence relations are solved and the result is $\mathcal{D}_n^{\texttt{join}} = n + 1$.

Let us consider again the definition of `join_tf/2` on page 47:

$$\begin{array}{lcl}
\texttt{join\_tf(P,Q)} & \xrightarrow{\alpha} & \texttt{join(P,Q,[]).}\\
\texttt{join(\ \ \ [],Q,\ \ \ [])} & \xrightarrow{\beta} & \texttt{Q;}\\
\texttt{join(\ \ \ [],Q,[I|A])} & \xrightarrow{\gamma} & \texttt{join([],[I|Q],A);}\\
\texttt{join([I|P],Q,\ \ \ \ A)} & \xrightarrow{\delta} & \texttt{join(P,Q,[I|A]).}
\end{array}$$

It was named `join_tf` because it is a version of `join` which is in **t**ail **f**orm. Clause γ pushes item `I` on top of `Q` but the actual contents of `Q` is never used. So let us find $\mathcal{D}_n^{\texttt{join\_tf}}$, that is, the delay of `join_tf(`P`,`Q`)`, where n is the length of list P. Instead of deducing a set of recurrence equations and solve them, let us endeavour with a direct approach consisting in counting, for each clause, how many times it is used:

- clause α is used once;
- clause δ is used n times (the first argument is reversed on the accumulator `A`),
- clause γ is used n times (the accumulator is empty at the beginning and the first list was reversed on it, so it contains n items when clause η is used),
- clause β is used once.

Note that the order of the clauses in the previous list follows the order of computation, for example, clause β is always used last. This allows us to express compactly this execution trace as the composition $\alpha\delta^n\gamma^n\beta$, where order matters. The total number of steps is $|\alpha\delta^n\gamma^n\beta| = |\alpha| + n \cdot |\delta| + n \cdot |\gamma| + |\beta| = 2n + 2 = \mathcal{D}_n^{\texttt{join\_tf}}$. We can now compare $\mathcal{D}_n^{\texttt{join}}$ and $\mathcal{D}_n^{\texttt{join\_tf}}$:

$$\mathcal{D}_n^{\texttt{join\_tf}} = 2 \cdot \mathcal{D}_n^{\texttt{join}}.$$

In other words, the version in tail form is twice as slow as the other, *even if we do not know the wall-clock delay of none.*

List Reversal. Let us reconsider now the following definition of `srev/1`, and investigate its delay:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

First, let us go for a mathematical approach. For any expression L whose value is a list, the delay $[\![\texttt{srev}(L)]\!]$ is, by definition, the sum of the costs of rewriting L and the cost of rewriting the call with the corresponding value $\overline{L}$ as an argument:

$$[\![\texttt{srev}(L)]\!] := [\![L]\!] + [\![\texttt{srev}(\overline{L})]\!].$$

Considering each clause separately, we can therefore write the equations defining the delays:

$$\begin{aligned}
[\![\texttt{srev([])}]\!] &:= 1,\\
[\![\texttt{srev([H|T])}]\!] &:= 1 + [\![\texttt{join(srev(T),[H])}]\!]\\
&= 1 + ([\![\texttt{srev(T)}]\!] + [\![\texttt{join(}\overline{\texttt{srev(T)}}\texttt{,[H])}]\!]).
\end{aligned}$$

Let us define the delay of `srev/1` in terms of the input size as

$$\mathcal{D}_n^{\texttt{srev}} := [\![\texttt{srev}(L)]\!],$$

where n is the length of L. Using this definition and assuming that `T` is bound to a list containing p items, then the length of `srev(T)` is also p and the previous equations, for $p \geqslant 0$, are equivalent to

$$\begin{aligned}
&\mathcal{D}_0^{\texttt{srev}} = 1,\\
&\mathcal{D}_{p+1}^{\texttt{srev}} = 1 + \mathcal{D}_p^{\texttt{srev}} + \mathcal{D}_p^{\texttt{join}} = 1 + \mathcal{D}_p^{\texttt{srev}} + (p+1) = 2 + p + \mathcal{D}_p^{\texttt{srev}}.
\end{aligned}$$

We would rather like to express $\mathcal{D}_p^{\texttt{srev}}$ in terms of p only—in other words, we want to solve the equations. The technique consists in making up all the differences of two successive terms in the series and then summing up these differences, so that terms cancel pairwise:

$$\begin{aligned}
\mathcal{D}_{p+1}^{\texttt{srev}} - \mathcal{D}_p^{\texttt{srev}} &= 2 + p, &&\text{where } p > 0,\\
\sum_{p=0}^{n-1} \left(\mathcal{D}_{p+1}^{\texttt{srev}} - \mathcal{D}_p^{\texttt{srev}}\right) &= \sum_{p=0}^{n-1} (2 + p), &&\text{where } n > 0.
\end{aligned}$$

Let us change variable p into $p-2$ in the right-hand side:

$$\mathcal{D}_n^{\mathsf{srev}} - \mathcal{D}_0^{\mathsf{srev}} = \sum_{p=2}^{n+1} p \;\Leftrightarrow\; \mathcal{D}_n^{\mathsf{srev}} - 1 = \sum_{p=2}^{n+1} p \;\Leftrightarrow\; \mathcal{D}_n^{\mathsf{srev}} = \sum_{p=1}^{n+1} p.$$

Let us double each side of the equality:

$$2 \cdot \mathcal{D}_n^{\mathsf{srev}} = 2 \cdot \sum_{p=1}^{n+1} p = \sum_{p=1}^{n+1} p + \sum_{p=1}^{n+1} p.$$

Let us change the variable p into $n-p+2$ in the second sum:

$$\begin{aligned} 2 \cdot \mathcal{D}_n^{\mathsf{srev}} &= \sum_{p=1}^{n+1} p + \sum_{p=1}^{n+1} (n-p+2) = \sum_{p=1}^{n+1} (p + (n-p+2)) \\ &= \sum_{p=1}^{n+1} (n+2) = (n+1)(n+2). \\ \mathcal{D}_n^{\mathsf{srev}} &= (n+1)(n+2)/2, \text{ where } n > 0. \end{aligned}$$

Since that replacing n by 0 in the latter formula gives

$$\mathcal{D}_0^{\mathsf{srev}} = (0+1)(0+2)/2 = 1,$$

we can therefore gather the two cases into one as

$$\mathcal{D}_n^{\mathsf{srev}} = (n+1)(n+2)/2, \text{ where } n \geqslant 0. \tag{3.3}$$

As an example, the number of rewrites for the call `srev([3,2,1])` on page 55 was found to be 10 and our formula indeed reckons this number: since `len([3,2,1])` $\equiv$ `3`, we need $\mathcal{D}_3^{\mathsf{srev}} = (3+1)(3+2)/2 = 4 \cdot 5/2 = 10$.

Would a lightweight approach have been possible? Yes. What we need to do is to use ellipses when we do not want to detail too much, but we must make sure that we do not get sloppy either. Let us resume from the recurrence equations and change p into $p-1$:

$$\mathcal{D}_0^{\mathsf{srev}} = 1; \quad \mathcal{D}_p^{\mathsf{srev}} = 1 + p + \mathcal{D}_{p-1}^{\mathsf{srev}}, \text{ where } p > 0, \tag{3.4}$$

and simply write a few terms of the series until $p = n$:

$$\begin{aligned} \mathcal{D}_0^{\mathsf{srev}} &= 1, \\ \mathcal{D}_1^{\mathsf{srev}} &= 1 + 1 + \mathcal{D}_0^{\mathsf{srev}}, \\ \mathcal{D}_2^{\mathsf{srev}} &= 1 + 2 + \mathcal{D}_1^{\mathsf{srev}}, \\ &\vdots \\ \mathcal{D}_n^{\mathsf{srev}} &= 1 + n + \mathcal{D}_{n-1}^{\mathsf{srev}}, \qquad \text{where } n > 0. \end{aligned}$$

By looking at the left-hand sides we find the terms $\mathcal{D}_0^{\mathsf{srev}}$, $\mathcal{D}_1^{\mathsf{srev}}$, ..., $\mathcal{D}_n^{\mathsf{srev}}$, whereas, in the right-hand sides, we can see $\mathcal{D}_0^{\mathsf{srev}}$, $\mathcal{D}_1^{\mathsf{srev}}$, ...,

$\mathcal{D}^{\mathsf{srev}}_{n-1}$, *which are all present in the left-hand side.* This gives us the idea to sum all the equations into one whose left-hand side will be $\mathcal{D}^{\mathsf{srev}}_0+\mathcal{D}^{\mathsf{srev}}_1+\cdots+\mathcal{D}^{\mathsf{srev}}_n$ and the right-hand side $\mathcal{D}^{\mathsf{srev}}_0+\mathcal{D}^{\mathsf{srev}}_1+\cdots+\mathcal{D}^{\mathsf{srev}}_{n-1}$ can be subtracted on both sides. That is, the technique is based on

$$\mathcal{D}^{\mathsf{srev}}_0+\mathcal{D}^{\mathsf{srev}}_1+\cdots+\mathcal{D}^{\mathsf{srev}}_n = x+\mathcal{D}^{\mathsf{srev}}_0+\mathcal{D}^{\mathsf{srev}}_1+\cdots+\mathcal{D}^{\mathsf{srev}}_{n-1}$$

being simply equivalent to $\mathcal{D}^{\mathsf{srev}}_n = x$. We need to explicit x now. It is made of two parts, which can be seen as two columns in the list of equations: the first is $1+1+\cdots+1$ ($n+1$ times, since it starts at $\mathcal{D}^{\mathsf{srev}}_0$ and ends at $\mathcal{D}^{\mathsf{srev}}_n$) and the second is $1+2+\cdots+n$. The former is simply the value $n+1$, while the latter should be familiar by now. If not, here is another way to calculate it. Let $S_n = 1+2+\cdots+n$. By writing the sum from right to left, this is still $S_n = n+(n-1)+\cdots+1$. By adding side by side these two equalities yield $S_n+S_n = (1+n)+(2+(n-1))+\cdots+(n+1) = (1+n)\cdot n$, so $S_n = n(n+1)/2$. Finally, we can gather all these findings into the expected result

$$\mathcal{D}^{\mathsf{srev}}_n = (n+1)+n(n+1)/2 = (n+1)(n+2)/2,\ \text{ where } n > 0.$$

This formula can be extended to $n = 0$ since it implies $\mathcal{D}^{\mathsf{srev}}_0 = (0+1)(0+2)/2 = 1$, as expected. We further have

$$\mathcal{D}^{\mathsf{srev}}_n = \frac{1}{2}n^2+\frac{3}{2}n+1 \ \Rightarrow\ \frac{2}{n^2}\mathcal{D}^{\mathsf{srev}}_n = 1+\frac{3}{n}+\frac{2}{n^2} \to 1,\ \text{ as } n\to\infty.$$

By definition of $(\sim)$, this implies

$$\mathcal{D}^{\mathsf{srev}}_n \sim \frac{1}{2}n^2,\ \text{ as } n\to\infty.$$

When a function f satisfies $f(n) \sim an^2$ for some constant a and with $n\to\infty$, it is said to be *asymptotically quadratic*. In our example at hand, multiplying by 10 the size of a large input list will multiply at least by 50 the computation time.

Let us reconsider now the definition of `rev/1` on page 60:

```
rev(L)              -α-> rev_join(L,[]).
rev_join(   [],Q)   -β-> Q;
rev_join([I|P],Q)   -γ-> rev_join(P,[I|Q]).
```

and assess the number of steps to compute `rev(`L`)`, with L being an expression whose value is a list containing n items. Considering each clause separately, we can then write the equations defining the delays

$$\begin{aligned}[\![\texttt{rev(L)}]\!] &\overset{\alpha}{=} 1+[\![\texttt{rev\_join(L,[])}]\!],\\ [\![\texttt{rev\_join([],Q)}]\!] &\overset{\beta}{=} 1,\\ [\![\texttt{rev\_join([I|P],Q)}]\!] &\overset{\gamma}{=} 1+[\![\texttt{rev\_join(P,[I|Q])}]\!].\end{aligned}$$

Let us now define the delay in terms of the size of the input, that is, the argument. The function call `rev(`L`)` has one list as argument. The usual way to measure the size of a list is to consider the number of elements it contains, that is, its length. Here, the size of the input L is characterised by an integer n. The same measure fits `rev_join/2`. Let us define the delay of `rev/1` and `rev_join/2` in terms of the input size:

$$\mathcal{D}_n^{\mathsf{rev}} := [\![\mathtt{rev}(L)]\!]; \quad \mathcal{D}_{n,m}^{\mathsf{rev\_join}} := [\![\mathtt{rev\_join}(P,Q)]\!],$$

where n is the length of P and m is the length of Q. Using these definitions and assuming that `T` contains p items and `A` contains q items, then the previous equations are equivalent to the following system:

$$\mathcal{D}_n^{\mathsf{rev}} \overset{\alpha}{=} 1 + \mathcal{D}_{n,0}^{\mathsf{rev\_join}}; \quad \mathcal{D}_{0,q}^{\mathsf{rev\_join}} \overset{\beta}{=} 1, \quad \mathcal{D}_{p+1,q}^{\mathsf{rev\_join}} \overset{\gamma}{=} 1 + \mathcal{D}_{p,q+1}^{\mathsf{rev\_join}}. \quad (3.5)$$

We would rather like to express $\mathcal{D}_p^{\mathsf{rev}}$ (respectively $\mathcal{D}_{p,q}^{\mathsf{rev\_join}}$), in terms of p only (respectively p and q). The solution consists in using the telescoping technique we have seen before, that is, in making the differences of two successive terms in the series $(\mathcal{D}_{p,q}^{\mathsf{rev\_join}})_{p\in\mathbb{N}}$ and summing them:

$$\begin{array}{rl}
 & \boxed{\mathcal{D}_{p,q}^{\mathsf{rev\_join}}} - \mathcal{D}_{p-1,q+1}^{\mathsf{rev\_join}} = 1 \\
+ & \mathcal{D}_{p-1,q+1}^{\mathsf{rev\_join}} - \mathcal{D}_{p-2,q+2}^{\mathsf{rev\_join}} = 1 \\
+ & \vdots \\
+ & \mathcal{D}_{1,q+(p-1)}^{\mathsf{rev\_join}} - \boxed{\mathcal{D}_{0,q+p}^{\mathsf{rev\_join}}} = 1 \\
\hline
\Rightarrow & \boxed{\mathcal{D}_{p,q}^{\mathsf{rev\_join}}} - \boxed{\mathcal{D}_{0,q+p}^{\mathsf{rev\_join}}} = \underbrace{1+1+\cdots+1}_{p\text{ times}} = p \\
\Leftrightarrow & \mathcal{D}_{p,q}^{\mathsf{rev\_join}} - 1 = p \\
\Leftrightarrow & \mathcal{D}_{p,q}^{\mathsf{rev\_join}} = p + 1, \text{ where } p > 0.
\end{array}$$

Since that replacing p by 0 in the last formula leads to the expected

$$\mathcal{D}_{0,q}^{\mathsf{rev\_join}} = 0 + 1 = 1,$$

we can gather all the cases into just one formula:

$$\mathcal{D}_{n,m}^{\mathsf{rev\_join}} = n + 1, \text{ where } n, m \geqslant 0.$$

Clearly, m is useless because it is absent from the right-hand side. Then

$$\mathcal{D}_n^{\mathsf{rev\_join}} = n + 1 \sim n; \qquad \mathcal{D}_n^{\mathsf{rev}} = n + 2 \sim n, \text{ as } n \to \infty. \qquad (3.6)$$

When a function f satisfies $f(n) \sim an$ for some constant a and with $n \to \infty$, it is said to be *asymptotically linear*. In other words, the delay will be proportional to the size of the input, for large inputs; that is a much better than a quadratic relationship.

Actually, a little bit more linguistic argument would have lead us quickly to the result, by counting the number of times each clause is used when rewriting a non-empty list of n items:

- clause α is used once;
- clause γ is used once for each item in the input, that is, n times;
- clause β is used once,

that is, the execution trace is $\alpha\gamma^n\beta$ and the total number of rewrites is indeed $|\alpha\gamma^n\beta| = |\alpha| + n \cdot |\gamma| + |\beta| = n + 2$.

We can now compare `srev/1` and `rev/1` in terms of delays:

$$\mathcal{D}_n^{\mathsf{srev}} = \frac{(n+1)(n+2)}{2} = (n+1)\frac{\mathcal{D}_n^{\mathsf{rev}}}{2} = \frac{n+1}{2} \cdot \mathcal{D}_n^{\mathsf{rev}}.$$

It is clear how much more efficient is `rev/1` with respect to `srev/1` and this comparison does not rely on any assumption about the speed of the Erlang run-time, for example, the virtual machine, with respect to the wall-clock time. Again, let us hammer down that the efficiency of `rev/1` is *not* due to its definition being in tail form, but because the algorithm it implements is more efficient. That is why we did not rename it `rev_-tf/1` because we only do this to signal that a definition is the tail form version of another and, this is not the case of `rev/1` in regard to `srev/1` (whose name is now revealed as standing for ***s**low **rev**ersal*).

Counting the Pushes We said above that we do not count the delay incurred by pushing items on top of a list, that is, we suppose

$$[\![\texttt{[I|L]}]\!] = [\![\texttt{I}]\!] + [\![\texttt{L}]\!],$$

because the construction of the value `[I|L]` is not the result of a user-defined function. We cannot compare or mix delays of predefined and user-defined functions because their implementation most probably rely on different execution mechanisms. But it is possible to make a complementary analysis of some user-defined function in order to count the number of items that are pushed on a list, for instance. Usually, this is done using the same approach we used for counting the number of rewrites to reach the results. For example,

```
srev(   [])   -> [];                               % No push
srev([I|L])   -> join(srev(L),[I]).           % At least one
join(   [],Q) -> Q;                                % No push
join([I|P],Q) -> [I|join(P,Q)].               % At least one
```

Assuming that `P` contains p items, we deduce the recurrence equations

$$\mathcal{P}_0^{\mathsf{srev}} = 0, \qquad\qquad \mathcal{P}_0^{\mathsf{join}} = 0,$$

$$\mathcal{P}^{\mathsf{srev}}_{p+1} = 1 + \mathcal{P}^{\mathsf{srev}}_{p} + \mathcal{P}^{\mathsf{join}}_{p}, \qquad \mathcal{P}^{\mathsf{join}}_{p+1} = 1 + \mathcal{P}^{\mathsf{join}}_{p},$$

where $\mathcal{P}^{\mathsf{srev}}_{p}$ and $\mathcal{P}^{\mathsf{join}}_{p}$ are the number of pushes needed to compute, respectively, `srev(`L`)` and `join(`P`,`Q`)`, with L, P and Q denoting lists and L and P contain p items each. The technique seen before lead to the solutions, for all $n \geqslant 0$,

$$\mathcal{P}^{\mathsf{srev}}_{n} = n(n+1)/2, \qquad \mathcal{P}^{\mathsf{join}}_{n} = n.$$

It is meaningless to consider delays of the form $\mathcal{D}^{\mathsf{srev}}_{n} + \mathcal{P}^{\mathsf{srev}}_{n}$ since there is no reason a priori that the time for processing one rewrite is the same as the time to process one push. Only the pair $(\mathcal{D}^{\mathsf{srev}}_{n}, \mathcal{P}^{\mathsf{srev}}_{n})$ is meaningful and could be used to compare two functions.

Exercises. [Answers on page 371.]

1. Find the delay for `rev_ter/1` defined as

```
rev_ter(   []) -> [];
rev_ter([I|L]) -> join_tf(rev_ter(L),[I]).

join_tf(P,Q)        -> join(P,Q,[]).
join(   [],Q,   []) -> Q;
join(   [],Q,[I|A]) -> join([],[I|Q],A);
join([I|P],Q,    A) -> join(P,Q,[I|A]).
```

2. Assess the maximum memory usage, including the call stack, of the previous functions. Justify your answer as precisely as you can.
3. Which of the two following equivalent definitions is the most efficient in terms of delay?

```
join_3a(P,Q,R) -> join(P,join(Q,R)).
join_3b(P,Q,R) -> join(join(P,Q),R).
```

Chapter 4

Filtering out

First occurrence. Let us consider the definition of a function `rm_-fst/2` such that the call `rm_fst(`I`,`L`)` is rewritten into L if I does not belong to the list L, otherwise it is rewritten to a list identical to L but without the first occurrence of I. This is our *specification*. For instance, we expect the following rewrites:

$$
\begin{aligned}
\texttt{rm\_fst(3,[])} &\twoheadrightarrow \texttt{[]};\\
\texttt{rm\_fst([],[])} &\twoheadrightarrow \texttt{[]};\\
\texttt{rm\_fst([5,2],[3,[]])} &\twoheadrightarrow \texttt{[3,[]]};\\
\texttt{rm\_fst(4,[[],[1,2],4,[],4])} &\twoheadrightarrow \texttt{[[],[1,2],[],4]};\\
\texttt{rm\_fst([],[4,[1,2],[],[],4])} &\twoheadrightarrow \texttt{[4,[1,2],[],4]}.
\end{aligned}
$$

First attempt. Let us try a direct approach. In particular, at this point, it is important *not* to seek a definition in tail form. Tail form must be considered as an optimisation and early optimisation is the root of all evil. The first idea that may come to mind is to define an auxiliary function `mem/2` such that the call `mem(`I`,`L`)` checks whether a given item I is in a given list L, because that notion of membership appears in the wording of the specification. But two problems would arise. Firstly, what would be the result of such a function? Secondly, what would be the added delay to use it? For the sake of the argument, let us try this line of thought. A list can either be empty or not, so let us make two clauses:

```
mem(I,   []) -> ☐;
mem(I,[J|L]) -> ☐.
```

Note that we introduced a variable `J`, distinct from variable `I`. *Two different variables may or may not denote the same value, but two occurrences of the same variable must denote the same value.* Had we written instead

```
mem(I,   []) -> □;
mem(I,[I|L]) -> □.          % Error: two occurrences of I
```

one case would be missing, namely when the head of the list is not the item sought for, for instance, `mem(3,[4])` would fail due to a match failure. Now, what is the first body? The first head matches if the list is empty. In particular, this means that the item is not in the list, since, by definition, an empty list is a list containing no item. How do we signify that? Since the original problem is silent on the matter, it is said to be *underspecified*. We may think that zero would be a token of choice to denote the absence of the item in the list:

```
mem(I,   []) -> 0;
mem(I,[J|L]) -> □.
```

But this would be a mistake because there is no natural relationship between the concept of emptiness and the number zero. (Why not 7?) Zero is best understood algebraically as the number noted 0 such that $0 + n = n + 0 = n$, for any number n. Then, let us try the empty list:

```
mem(I,   []) -> [];                    % I is not reused
mem(I,[J|L]) -> □.
```

The next step is to find a way to actually compare the value of `I` to the value of `J`. We can use the rule above about variables: two occurrences of the same variable mean that they hold the same value. Therefore

```
mem(I,   []) -> [];
mem(I,[I|L]) -> □.
```

was not so bad, after all? Indeed, but we know now that a case is missing, so let us add it at the end:

```
mem(I,   []) -> [];
mem(I,[I|L]) -> □;
mem(I,[J|L]) -> □.                              % I ≠ J
```

Since the run-time, for instance, a virtual machine, of Erlang scans the heads top-down, if the last head matches the current call, it means that `I` and `J` denote different values. Now, what is the second body? It stands after we found that the item we were looking for is present at the head of the list, thus it belongs to the list. How do we signify that? We may think of ending with the item itself, the rationale being that if the result is the empty list, then the item is not in the input list, otherwise the result is the item itself (and it is present in the input list):

```
mem(I,   []) -> [];
mem(I,[I|L]) -> I;                         % Ignoring L
```

`mem(I,[J|L]) ->` $\boxed{}$.

The last body is easier to guess since it deals with the case where the head of the list (`J`) is not the item we seek (`I`), so a recursive call which ignores `J` should come to mind:

```
mem(I,   []) -> [];
mem(I,[I|L]) -> I;
mem(I,[J|L]) -> mem(I,L).                          % Ignoring J
```

We could even simplify this definition by replacing variables in the heads where they occur only once and which do not occur in their corresponding bodies (that is, their value is ignored) with an underscore:

```
mem(_,   []) -> [];                                   % Here,
mem(I,[I|_]) -> I;                                     % here
mem(I,[_|L]) -> mem(I,L).                          % and here
```

That is the reason why we choose the names of the auxiliary functions, that is, functions that are not expected to be used by the user directly (and are not exported outside their module), to end with two underscores: it is a syntactical error in Erlang to have a pattern containing two successive underscores standing for parameters and we sometimes use one underscore at a time in the names, so two underscores avoid any confusion.

Some tests would increase the confidence that this definition is correct and complete with respect to the specification. Let us label the clauses first:

$$\begin{aligned}&\texttt{mem(\_,\ \ \ [])} \xrightarrow{\zeta} \texttt{[];}\\ &\texttt{mem(I,[I|\_])} \xrightarrow{\eta} \texttt{I;}\\ &\texttt{mem(I,[\_|L])} \xrightarrow{\theta} \texttt{mem(I,L).}\end{aligned}$$

Then we could try the following cases:

$$\begin{aligned}\texttt{mem(3,[])} &\xrightarrow{\zeta} \texttt{[],}\\ \texttt{mem(3,[1])} &\xrightarrow{\theta} \texttt{mem(3,[])} &&\xrightarrow{\zeta} \texttt{[],}\\ \texttt{mem(3,[1,3,2])} &\xrightarrow{\theta} \texttt{mem(3,[3,2])} &&\xrightarrow{\eta} \texttt{3.}\end{aligned}$$

The code seems to work: item `I` is in list `L` if the result is `I`, otherwise it is `[]`. However, the program is not correct. The hidden and flawed assumption that lead to the error is “items can not be lists,” in spite of the examples given for illustrating the expected behaviour of `rm_fst/2`, which contained cases where the item was a list. In particular, an item can be the empty list and this situation leads to an ambiguity with our definition of `mem/2`:

$$\texttt{mem([],[])} \xrightarrow{\zeta} \texttt{[]},\quad \texttt{mem([],[[]])} \xrightarrow{\eta} \texttt{[]}.$$

It is impossible to distinguish the two cases, the first one meaning absence and the second presence of the item, because they both end with the empty list. This problem could have been discovered earlier or during testing. Empirical evidence advises to test a given definition with the empty list in as many locations as possible in the input. If we wish to maintain the polymorphism of the solution to our problem, that is, if we really want the definition to accept lists of any kind of items, there is no way out. Actually, the definition we came up with is correct if the items are all integers, but we have no way yet to implement this restriction. We shall indeed meet other kinds of values later, making this simple question of membership straightforward to answer. But, for now, let us backtrack and ask ourselves whether using `mem/2`, assuming we have it, is really a good idea.

Better approach. Let us suppose that the input list contains the item at the end. Using `mem/2` to find it leads to a complete traversal of the input list. Then another traversal from the beginning (the head of the list) is needed to make a new list without the last item, so, in total, two traversals are performed. But a better idea consists in *interleaving* these two passes into one because the problem stems from the fact that `mem/2` forgets about the items which are not the item of interest, thus, when it is found or known to be absent, there is no way to remember the past items in order to make the result. By interleaving, we mean that during the traversal, the concepts of membership and of "rebuilding a list without an item" are combined, instead of being used separately as two function calls. A similar situation was encountered in the design of a function reversing a list: the version which made use of a *function* to join two lists was less efficient than the version which included the *concept* of joining. The algorithm consists in memorising all visited items, so they are not forgotten and, if the item is not found, the resulting list is rebuilt from them; if found, the result is built from them *and* the remaining, unvisited, items. There are usually two ways to implement this auxiliary memory of visited items: either by using an accumulative parameter, called *accumulator*, or, quite simply, by means of the control context of a recursive call. At this point, it is important to recall a cardinal guideline: Do not try first to design a definition in tail form, but opt instead for a direct approach. Once a correct and complete definition has been reached, if the data set is too big to fit in the call stack as allocated by the operating system, then a tail form definition should be obtained by transforming the current definition. In some simple cases, a direct approach can be in tail form, but the point still holds in general: at first, let us ignore all concerns about the definition being in tail form. Accordingly, let us use the control context of a recursive call to record the visited items. A list being either empty or not, the following two clauses come naturally to mind:

```
rm_fst(I,   []) -> ▭;
rm_fst(I,[J|L]) -> ▭.
```

Then, just as we tried with `mem/2`, we must distinguish the case when `I` is the same as `J`:

```
rm_fst(I,   []) -> ▭;
rm_fst(I,[I|L]) -> ▭;           % Equality constraint
rm_fst(I,[J|L]) -> ▭.
```

Now, we know that the last clause deals with the case when the item `I` is not `J`, thus we must memorise `J` and go on comparing `I` with the other items in `L` (if any). This is where the recursive call with a control context, discussed above, is set as `[J|␣]`:

```
rm_fst(I,   []) -> ▭;
rm_fst(I,[I|L]) -> ▭;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].
```

Very importantly, let us remark that the position of `J` in the result is the same as in the input (the head of a list). The second clause corresponds to the case where the item we are looking for, namely `I`, is found to be the head of the current list, which is a sub-list of the original input. A list made of successive items from the beginning of a given list is called a *prefix* of the latter. When a list is a sub-list of another, that is, it is made of successive items including the last, it is called a *suffix*. We know that the `I` in `[I|L]` is the first occurrence of `I` in the original list (the one in the first call), because we wouldn't be dealing with this case *again*: the specification states that this first occurrence must be absent from the resulting list; since it is now at the head of a suffix, we just need to end with `L`, *which we do not visit*:

```
rm_fst(I,   []) -> ▭;
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].
```

The first clause handles the case where we traversed the whole original list (up to `[]`) without finding `I`. Thus the result is simply the empty list because the empty list without `I` is the empty list:

```
rm_fst(I,   []) -> [];
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].
```

We can simplify a little bit by muting variables whose values are useless and finally settle for

```
rm_fst(_,   []) -> [];                                % Here
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].
```

Note that it is incorrect to write

```
rm_fst(_,   []) -> [];
rm_fst(_,[_|L]) -> L;                                % Error
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].         % Becomes useless
```

We must understand the underscore (_) as an unknown variable, yet unique in the pattern where it occurs. So two underscores in the same pattern, as we mistakenly just put, may or *may not* denote the same value. Only variables which are not used in the body *and* which are not repeated in the head can be correctly replaced by underscores. Perhaps is it timely now to comment on the name we chose: `rm_fst`. It obviously stands for "remove (the) first (occurrence)," but, as we came to realise, nothing is actually removed from the original list: every time a new list is reconstructed from it, but without the first occurrence of a given item. The usual conception of removing supposes a permanent state, an object or entity of some sort, such that, after the removal, some part of it is lacking. This idea has no avatar in functional languages, like Erlang, whose run-time environments build new values from old ones only by copying and sharing. This, of course, bears the question of how and when is old data discarded by the run-time. This task is appointed to the *garbage collector*, which can be thought of as a concurrent process with access to the memory of the program being executed. There are many kinds of strategies to determine when some data is definitely useless and we shall not expand on this complicated subject here. Just let us imagine that there is an invisible oracle which is taking care of the deallocation of useless data, that is, releasing memory space for other usages. The second question that is also logically entailed is how costly is it to copy parts from the old data, for instance, the arguments of a function call, to the new data, for instance, the result of a function call. For example, in the definition of `rm_fst/2`, the second body is `L`, which is a suffix of the original (input) list. Is it necessarily duplicated in memory or can it be shared between the input and the output? We shall answer this question later.

Let us run some tests now and, in order to follow them easily, it is handy to label the clauses with some Greek letters:

```
rm_fst(_,   []) $\xrightarrow{\alpha}$ [];
```

`rm_fst(I,[I|L])` $\xrightarrow{\beta}$ `L;`
`rm_fst(I,[J|L])` $\xrightarrow{\gamma}$ `[J|rm_fst(I,L)].`

We can then try some of the examples given on page 75.

`rm_fst([],[4,[1,2],[],[],4])`
$\xrightarrow{\gamma}$ `[4|rm_fst([],[[1,2],[],[],4])]`
$\xrightarrow{\gamma}$ `[4|[[1,2]|rm_fst([],[[],[],4])]]`
$=$ `[4,[1,2]|rm_fst([],[[],[],4])]`
$\xrightarrow{\beta}$ `[4,[1,2]|[[],4]]`
$=$ `[4,[1,2],[],4].`
`rm_fst([5,2],[3,[]])` $\xrightarrow{\gamma}$ `[3|rm_fst([5,2],[[]])]`
$\xrightarrow{\gamma}$ `[3|[[]|rm_fst([5,2],[])]]`
$=$ `[3,[]|rm_fst([5,2],[])]`
$\xrightarrow{\alpha}$ `[3,[]|[]]`
$=$ `[3,[]].`

Once we are convinced that our definition is correct and complete with respect to the specification, there is a little extra worth testing: we can check *what happens for inputs which are not expected by the specification.* Our specification says at one point that the second argument of `rm_fst/2` is a list. What happens if we supply an integer instead? For example, we have `rm_fst([],3)` $\not\rightarrow$. We have a match failure, that is, the rewrites are stuck, which means that our definition is not *robust*, that is, it fails abruptly on unspecified inputs. When programming in the small, as we do here, robustness is usually not a concern because we want to focus on learning the language and how to express simple algorithms with it, but when developing programs whose customers are other people or simply when making large applications, even for ourselves, it is important to make the code robust by means of error signalling, catching and display. Notice that a program can be complete but not robust, because completeness is relative to what is specified (all valid inputs must be accepted and not lead to an error), whilst robustness is relative to what is left unspecified. In the present case, what can be done to add robustness to our definition of `rm_fst/2`? We can add a *last* clause to handling the case where the second argument is not a list and make the body be the empty list:

```
rm_fst(_,   []) -> [];
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)];
rm_fst(_,    _) -> [].                         % A catch-all clause
```

But why the empty list in the first place? Why not? Any kind of list would be fine, actually. The first clause has become useless, because the empty list is matched by the underscore, as the underscore is a special variable and variables match anything, so we can simplify the definition:

```
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)];
rm_fst(_,    _) -> [].
```

Now rm_fst/2 never fails but Erlang does not enforce that this function will never be called with a second argument which is not a list. Other functional languages, like Haskell or OCaml, feature *type systems* that enable their compilers to prove that all calls are *type-safe*, that is, their computations will never lead to an error due to the nature of some argument or, generally, to values being used with wrong assumptions, for instance, they forbid expressions like `[] + 1`, which is allowed by Erlang compilers. In OCaml, the question of the robustness of the definition of rm_fst__/3 would not be raised, since the compiler would refuse to compile a program with a call to it with a second argument that would not be a list. Thus there would be no need and even no way to add a catch-all clause as we did here. It may be argued, of course, that the caller will not know that it called rm_fst/2 with a wrong second argument. These considerations are akin to discussing the merits and weaknesses of scripting languages, which try very hard to ignore errors by defaulting on special values (like the empty string) to keep running, as opposed to programming languages that impose strong requirements at compile-time about the way values are processed. If defaulting on the empty list is considered an issue, Erlang allows another kind of value, in addition to integer and lists: *atoms*. Atoms are enumerated values whose name only is known. In other words, an atom is known by its name and the fact that it is unique in the whole program, that is, another atom with a different name is a different value. By contrast, two different variables can denote the same value. Atoms follow the same lexical rules as function names, that is: they must start with a lowercase letter and may be followed by any number of letters, numbers or underscores. For instance error and empty and hello are valid atoms and we can trust that they are pairwise different, that is, their unknown values are distinct when considered two by two: error $\neq$ empty etc.

Here is the definition of a function which distinguishes between lists and non-list arguments:

```
is_a_list(   []) -> yes;
is_a_list([_|_]) -> yes;
is_a_list(    _) -> no.
```

As any other kind of value, atoms can be mixed within more complex values, for instance, `[8,blue,yellow,17,green]`. Atoms are handy to signal errors because they are unique identifiers, therefore they cannot be confused with any other kind of data the function computes and so can be detected easily by the caller. Consider this robust version of `rm_fst/2` which distinguishes errors from specified computations:

```
rm_fst(_,   []) -> [];
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)];
rm_fst(_,    _) -> error.                           % An atom
```

Then a function calling `rm_fst/2` can make the difference between a normal rewrite and an error by using an atom as a pattern:

```
caller(I,L)   -> check(rm_fst(I,L)).
check(error)  -> □;                        % Atom as a pattern
check(Result) -> □.
```

In order to develop further our programming skills, let us imagine that a tail form definition of `rm_fst/2` is needed. The best is to start from the version not in tail form and see where it fails to be in tail form. Only one body is not in tail form: the last, precisely because we use the control context to store the items which are not the one we are looking for. We hence need to add an accumulative parameter `A` to the definition, which becomes `rm_fst/3`, and we define a new function `rm_-fst_tf/2` which must be, in the end, equivalent to `rm_fst/2` but in tail form (we frame the code to be reconsidered and in bold is the change):

```
rm_fst_tf(I,L)    -> rm_fst(I,L,□).
rm_fst(_,   [],A) -> [];
rm_fst(I,[I|L],A) -> L;
rm_fst(I,[J|L],A) -> [J|rm_fst_tf(I,L,A)].
```

We learnt from the transformation from `join/2` to `join_tf/2`, on pages 45–47, that we can push the variable in the control context, that is, `J`, onto the accumulator `A`:

```
rm_fst_tf(I,L)    -> rm_fst(I,L,□).
rm_fst(_,   [],A) -> [];
rm_fst(I,[I|L],A) -> L;
rm_fst(I,[J|L],A) -> rm_fst_tf(I,L,[J|A]).
```

Now `J` is pushed on `A`, not on the call anymore, therefore the order of the items is reversed: the second argument is reversed on the accumulator until either the end is reached or `I` is found. If the latter (second clause of `rm_fst/3`), we cannot just return `L` as we did because there is no control context to provide the past items anymore: we need to use the accumulator. We already know that `A` contains the previous items in reverse order and `L` contains the remaining, unvisited, items. So we can try the following:

```
rm_fst_tf(I,L)    -> rm_fst(I,L,□).
rm_fst(_,   [],A) -> [];
rm_fst(I,[I|L],A) -> join(rev(A),L);
rm_fst(I,[J|L],A) -> rm_fst_tf(I,L,[J|A]).
```

At this point, we should hear a bell ringing because this body reminds us of `srev/1` defined as

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

and about which we determined that its delay was asymptotically quadratic; in other words, it is slow. The reason of its slowness was due to the repeated calls to `join/2` with the length of the first argument ranging from $n-1$ to 0, where n is the length of the input list. This deficiency was remedied by using an auxiliary function `rev_join/2` with an accumulator:

```
rev(L)            -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

A look back at FIGURE 13 on page 56 suggests that if the input list has length n, then the first item is pushed $n-1$ times in total, the second item $n-2$ etc. until the last item is unmoved. So the problem boiled down to the reprocessing the same data over and over again, not to the function `join/2` being slow or evil in itself.

Improvements. Then what should we do to improve on this slow scheme? The study of the previous definition gives the key. The purpose is to reverse a list (`A`) on top of another (`L`). This is what `rev_join/2` *efficiently* does:

```
rm_fst_tf(I,L)    -> rm_fst(I,L,□).
rm_fst(_,   [],A) -> [];
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst_tf(I,L,[J|A]).
```

It is incorrect to write `rev_join(L,A)` in stead of `rev_join(A,L)`. We can now finish the definition because we know exactly the contents of the accumulator: it contains a reversed prefix of the list up to the current situation. In the case of the second clause, the situation in question is that the end of the original list has been reached (`[]`). In other words, `A` contains the complete original list in reverse order. Hence, the complete definition is

```
rm_fst_tf(I,L)     -> rm_fst(I,L,□).
rm_fst(_,   [],A) -> rev(A);
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst_tf(I,L,[J|A]).
```

The work is not done yet, as the initial value of the accumulator has to be determined. (As a side note, this example shows that a program is not compulsorily best written from top to bottom. After all, programs are not English essays and programming is perhaps more akin to painting.) Since it is a list in which another is being reversed to be later reversed again, it is important to start with an empty list, otherwise extraneous initial items would be found in the final value:

```
rm_fst_tf(I,L)     -> rm_fst(I,L,[]).
rm_fst(_,   [],A) -> rev(A);
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst_tf(I,L,[J|A]).
```

Compiling this definition, which needs first to be embedded in a module, displays an error:

```
function rm_fst_tf/3 undefined
```

in the last clause. Indeed, it should be `rm_fst` instead of `rm_fst_tf`. So the correct tail form definition is

```
rm_fst_tf(I,L)     -> rm_fst(I,L,[]).
rm_fst(_,   [],A) -> rev(A);
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst(I,L,[J|A]).              % Fixed
```

More precisely, `rm_fst_tf/2` is in tail form because the definition of a function is in tail form if, and only if,

1. the bodies of its definition are in tail form. A body is in tail form if it is a constant (either a variable, an integer, an atom or a list made of these or empty) or a function call whose arguments are either constants or arithmetic expressions.
2. the functions called in these bodies are in tail form too.

The definition of "being in tail form" is syntactical in nature, that is, it depends only on the shape of the code, not on what it computes, and it is recursive. The definition of the function itself does not need to be recursive, since, for example, `id(X) -> X.` is a definition in tail form which is not recursive. For example, if `rev/1` were not in tail form, then `rm_fst/3` would not be in tail form as well, despite having the right shape (no control contexts for the calls) and so would not `rm_fst_tf/2` be in tail form.

Membership test. Before analysing the performance of `rm_fst/2` and `rm_fst_tf/2`, let us backtrack and find a definition to the restive `mem/2`. The key is to use some atom to distinguish the case when the sought item is present and when it is absent. Let us use two atoms, one for each case:

```
mem(_,   []) -> false;
mem(I,[I|_]) -> true;
mem(I,[_|L]) -> mem(I,L).
```

Note that the choice of the atom, just like a variable, is important for the legibility of the program. It is important not to mislead the caller by writing, for instance

```
mem(_,   []) -> true;
mem(I,[I|_]) -> false;
mem(I,[_|L]) -> mem(I,L).
```

despite this being a valid definition. Anyhow, whatever the atoms, the specification and the comments should state their interpretation explicitly. Here, we would expect something like: “The function `mem/2` determines the membership of an item to a list. The function call `mem(`I`,`L`)` results in the atom `true` if, and only if, item I is an element of list L, otherwise the result is the atom `false`. If L is not a list, the call fails immediately due to a match failure.” We insist in using slanted letters, for instance, I or L, instead of `I` or `L`, because I, for example, denotes *any* Erlang value whatsoever, while `I` denotes any Erlang value *in a specific clause*. Technically, I and L are called *metavariables* because they denote Erlang expressions independently of any particular definition, as opposed to variables which denote a value in a particular clause of a particular definition.

On a last note, we should underline that the multiple occurrences of `I` in the definition of `mem/2` implies an equality test at run-time that can be slow if `I` denotes a large value. However, for the sake of simplicity, we will not take this kind of implicit computational overhead into account when assessing the speed of a function. As a rough approximation of the wall-clock time, we shall only count the number of functions calls needed to evaluate a given function call, what we call its *delay*. As an extreme example demonstrating that this is not the whole story, mind

```
tautology(X) -> eq(X,X).
```

```
eq(X,X) -> true;
eq(_,_) -> false.
```

Delays. Let us assess the delay of `rm_fst/2` and `rm_fst_tf/2`. Contrary to previous functions up to now, the delay of `rm_fst(`I`,`L`)` depends on the position of the first occurrence of item I in the list L, of length n, because of clause β in

```
rm_fst(_,   [])  -α-> [];                              % Absent
rm_fst(I,[I|L])  -β-> L;                               % Present
rm_fst(I,[J|L])  -γ-> [J|rm_fst(I,L)].          % Search further
```

More precisely, there are three cases:

- L is empty;
- L is not empty and I is absent;
- L is not empty and I occurs at position k, counting the head of L as position 0.

In the first case, the delay is 1, by clause α. Otherwise, the recursive call in clause γ is repeated n times until either clause α or β applies. If the item is absent, then α applies and the delay is $n+1$. Otherwise, it is $k+1$. All these execution traces can be formally summarised as $\gamma^n\alpha + \gamma^k\beta$, where the addition signifies a disjunction, that is, in operational terms, a choice. (Composition of clauses do not commute, for instance, $\gamma\alpha \neq \alpha\gamma$, but disjunction does, for instance, $\alpha+\beta = \beta+\alpha$.) For example, the delay of `rm_fst(1,[4,2,1,7,1])` is 3 because clause γ is used 2 times and, finally, clause β once, which corresponds to the execution trace $\gamma^2\beta$. If we interpret $k = n$ as meaning that I is absent from a non-empty L, then the delay is always $k + 1$, which is concisely expressed by $|\gamma^k(\alpha+\beta)| = k \cdot |\gamma| + |\alpha+\beta| = k+1$. The length of a disjunction of clauses, for instance, $|\alpha + \beta|$, is defined only if the disjoint sub-traces have the same length, here, $|\alpha| = |\beta| = 1$. We cannot extend the formula $k + 1$ to be valid when L is empty, as it would require $k = 0$ and this case means that I is the head of L. If we note $\mathcal{D}_{n,k}^{\texttt{rm\_fst}}$ the delay for computing the call `rm_fst(`I`,`L`)`, where L contains n items and k is the position at which the first occurrence of I occurs in L, then

$$\mathcal{D}_{n,k}^{\texttt{rm\_fst}} = k + 1, \text{ with } 0 \leqslant k \leqslant n,$$

with the convention that $k = n$ if it is missing. If the list is empty, that is, $n = 0$, then the inequalities imply that $k = 0$, but this is not interpreted as I being the head of L.

It is now easy to deduce what are the best and worst cases for the delay, that is, for a given size n of the input, what are the configurations that lead to a minimum or maximum number of rewrites to reach the result, that is, a value.

Let us note $\mathcal{B}_n^{\texttt{rm\_fst}}$ the delay in the best case and $\mathcal{W}_n^{\texttt{rm\_fst}}$ the delay in the worst case. The above formulas define them immediately. The minimum value of $\mathcal{D}_{n,k}^{\texttt{rm\_fst}}$ is when $k = 0$ (keep in mind that n is fixed in this analysis), that is, the item we sought is the head of the list. If the list is empty, that is, $n = 0$, then there is only one case and the delay is 1. The worst case happens when $k = n$, that is, when the item is absent from the list. For $n \geqslant 0$,

$$\mathcal{B}_n^{\texttt{rm\_fst}} := \min_{0 \leqslant k \leqslant n} \mathcal{D}_{n,k}^{\texttt{rm\_fst}} = 1,$$
$$\mathcal{W}_n^{\texttt{rm\_fst}} := \max_{0 \leqslant k \leqslant n} \mathcal{D}_{n,k}^{\texttt{rm\_fst}} = n + 1 \sim n, \text{ as } n \to \infty.$$

The best and worst delays provide bounds for the delay, but more can be said. The *average delay* is the arithmetic mean of the delays on all possible inputs. Usually, the number of inputs is infinite so some additional assumptions are usually necessary, depending on the problem at hand. In the case of `rm_fst/2`, the only thing that matters is the position of the item to be removed and it does not suppose anything about the other items, for instance, their relative order or value. The problem is thus equivalent to being given a list L and considering all its items for removal. This leads to define the average delay as

$$\mathcal{A}_n^{\texttt{rm\_fst}} := \frac{1}{n} \sum_{I \in L} [\![\texttt{rm\_fst(}I\texttt{,}L\texttt{)}]\!].$$

Consider that the average delay is not the measure of the delay of a single operation because it may be that no configuration of the input leads to the average delay. It may also not be the most probable delay of a single operation, because this depends on the distribution of all the delays. Let us resume our calculation. Since an item in a list is uniquely referred to by its position, it is equivalent to write

$$\mathcal{A}_n^{\texttt{rm\_fst}} = \frac{1}{n} \sum_{k=0}^{n-1} \mathcal{D}_{n,k}^{\texttt{rm\_fst}} = \frac{1}{n} \sum_{k=0}^{n-1} (k+1) = \frac{1}{n} \sum_{k=1}^{n} k = \frac{n+1}{2}. \tag{4.7}$$

We cannot compare $\mathcal{A}_n^{\texttt{rm\_fst}}$ and $\mathcal{W}_n^{\texttt{rm\_fst}}$ because the latter assumed that the item is absent, whilst the former assumed the contrary. The worst case when the item is present happens when the item is last, that is, $k = n - 1$, the delay is then $\mathcal{W}_n^{\texttt{rm\_fst}} = \mathcal{D}_{n,n-1}^{\texttt{rm\_fst}} = (n-1) + 1 = n$. Therefore,

$$\mathcal{A}_n^{\texttt{rm\_fst}} \sim \tfrac{1}{2} \mathcal{W}_n^{\texttt{rm\_fst}}, \text{ as } n \to \infty. \tag{4.8}$$

In other words, the asymptotic average delay is midway between the best and worst delays (the best delay is 1).

For the sake of versatility, let us prove that $\sum_{k=1}^{n} k = n(n+1)/2$ in yet another way, by telescoping successive terms of the series $(n^2)_{n>0}$, that is, summing the differences $(n+1)^2 - n^2$. The key is that these differences have a simple, closed expression: $(n+1)^2 - n^2 = 2n+1$.

$$\begin{array}{rr}
 & (1+1)^2 - \boxed{1^2} = 2^2 - 1^2 = 2 \cdot 1 + 1 \\
+ & (2+1)^2 - 2^2 = 3^2 - 2^2 = 2 \cdot 2 + 1 \\
+ & \vdots \\
+ & \boxed{(n+1)^2} - n^2 = 2 \cdot n + 1 \\
\hline
\Rightarrow & \boxed{(n+1)^2} - \boxed{1^2} = 2\sum_{k=1}^{n} k + n.
\end{array}$$

Finally: $\sum_{k=1}^{n} k = n(n+1)/2$. Let us consider now the delay for calls `rm_fst_tf(`I`,`L`)`, with

$$\begin{array}{lcl}
\texttt{rm\_fst\_tf(I,L)} & \xrightarrow{\alpha} & \texttt{rm\_fst(I,L,[]).} \\
\texttt{rm\_fst(\_,\ \ \ [],A)} & \xrightarrow{\beta} & \texttt{rev(A);} \\
\texttt{rm\_fst(I,[I|L],A)} & \xrightarrow{\gamma} & \texttt{rev\_join(A,L);} \\
\texttt{rm\_fst(I,[J|L],A)} & \xrightarrow{\delta} & \texttt{rm\_fst(I,L,[J|A]).}
\end{array}$$

Just as before, there are three cases:

- L is empty;
- L is not empty and I is absent;
- L is not empty and I occurs at position k, counting the head of L as position 0.

In the first case, clause α is used once, then clause β once and finally we have to add the delay of the call `rev([])`, which is 2. The total delay is thus 4.

Let us now assume that L contains $n > 0$ items and that the item I we are looking for is missing from L. Then the call `rm_fst_tf(`I`,`L`)` leads to

- clause α being used once,
- clause δ being used n times,
- clause β being used once,
- clause γ being unused.

This execution trace is formally $\alpha\delta^n\beta$, but it is partial as we must take into account the delays of the calls other than to `rm_fst/3`: only `rev(A)` remains and we know that its delay is always the length of `A` plus 2 (on page 71). What is the length of the accumulator in the body of clause β? The body of clause δ gives the answer: the accumulator contains all the items from the input list but in reverse order, that is, its length is also n. Therefore the total delay is $(1) + (n) + (1 + (n + 2)) = 2n + 4$.

Now let us suppose that I does occur in L at position k, counting the head of the list as being at position 0. Then

- clause α is used once;
- clause δ is used k times, from positions 0 to $k - 1$;
- clause β is unused;
- clause γ is used once.

Formally, this is the partial execution trace $\alpha\delta^k\gamma$, which must be completed by taking into account the call `rev_join(A,L)`. We know from equation (3.6) on page 71 that $\mathcal{D}_n^{\texttt{rev\_join}} = n + 1$, where n is the length of the first argument. In clause γ this argument is `A` and we know that it contains the first k items of L, because clause δ was used k times. Therefore, we have to add $k + 1$ steps to the previous count, so the total delay is $1 + k + 1 + (k + 1) = 2k + 3$, with $k < n$.

We may wonder why there is a difference in the additive constants of the two last cases: $2n+4$ versus $2k+3$. Looking again at our calculations reveals that the difference comes from calling `rev(A)` in clause β, which in turn calls `rev_join(A,[])`: this adds one more rewrite, whereas we could call `rev_join/2` directly, just like in clause γ. It is in fact equivalent to define `rm_fst_tf/2` as follows (change in bold):

$$\begin{array}{lcl}
\texttt{rm\_fst\_tf(I,L)} & \xrightarrow{\alpha} & \texttt{rm\_fst(I,L,[]).}\\
\texttt{rm\_fst(\_,~~~[],A)} & \xrightarrow{\beta} & \textbf{\texttt{rev\_join(A,[]);}}\\
\texttt{rm\_fst(I,[I|L],A)} & \xrightarrow{\gamma} & \texttt{rev\_join(A,L);}\\
\texttt{rm\_fst(I,[J|L],A)} & \xrightarrow{\delta} & \texttt{rm\_fst(I,L,[J|A]).}
\end{array}$$

Now the delay when the item is absent becomes $2n+3$, mirroring nicely the delay when it first occurs at position k, that is, $2k + 3$. Note that making the previous change in order to save one function call is not something useful in general. We did it because it helped evince that there is a common concept shining through the different cases: if we take the convention that $k = n$ means that I is not in L, then the delay is $2k+3$ if L is not empty. If it is empty, the delay is $1+1+(0+1) = 3$, which would be obtained by taking $k = 0$ in $2k + 3$, but this would normally be interpreted as meaning that I is the first item of L, that is, the head of L. Hence the empty list seems to be a special case. If we really want to have one single formula for the delay, we can take the additional convention that $k = -1$ means that the list is empty. This leads to a delay of $2 \cdot (-1) + 3 = 1$. Can we modify the definition so that becomes true without changing the other cases? Yes, by adding a clause:

```
rm_fst_tf(I,[])   -> [];
rm_fst_tf(I, L)   -> rm_fst(I,L,[]).
rm_fst(_,   [],A) -> rev_join(A,[]);
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst(I,L,[J|A]).
```

Again, this last modification is only made with a didactic purpose in mind and it may not be worthwhile in general, but it does allow one to roundly say that

$$\mathcal{D}^{\mathsf{rm\_fst\_tf}}_{n,k} = 2k+3,\ \text{ where }\ -1 \leqslant k \leqslant n,$$

k being the position of I in L, whose length is n; $k = -1$ means that L is empty, so $n = 0$; $k = n$ means that I is absent from a non-empty L.

We can now deduce what are the best and worst cases, by respectively minimising and maximising the two previous delays when k varies—recall that the best and worst cases are defined for a given n. The best case is when $k = 0$ and the worst is when $k = n$. Thus, the best case happens when I is the head of L and the worst case is when I is absent from L. If the list is empty, that is, $k = -1$, it means that L is empty, that is, $n = 0$, and, thus, there is only one case which is both the best and the worst. If we note $\mathcal{W}^{\mathsf{rm\_fst\_tf}}_{n}$ the delay in the worst case and $\mathcal{B}^{\mathsf{rm\_fst\_tf}}_{n}$ the delay in the best case, we have, for $n > 0$,

$$\mathcal{D}^{\mathsf{rm\_fst\_tf}}_{0,-1} = \mathcal{B}^{\mathsf{rm\_fst\_tf}}_{0} = \mathcal{W}^{\mathsf{rm\_fst\_tf}}_{0} = 1,$$
$$\mathcal{B}^{\mathsf{rm\_fst\_tf}}_{n} = 3, \quad \mathcal{W}^{\mathsf{rm\_fst\_tf}}_{n} = 2n+3 \sim 2n,\ \text{ as } n \to \infty.$$

How does `rm_fst/2` and `rm_fst_tf/2` compare in terms of delays? When comparing delays, we must be cautious because, while it is always mathematically possible to relate real numbers, doing so is not necessarily meaningful. For instance, let us suppose that we have two functions `f/1` and `g/1` and we determined their exact delays $\mathcal{D}^{\mathsf{f}}_{n}$ and $\mathcal{D}^{\mathsf{g}}_{n}$. If both functions are equivalent, that is, if for all the same inputs they compute the same outputs, it is meaningful to compare their exact delays and also their asymptotic delays. If their delays are only known in some worst case, it is important to make sure that the worst case of `f/1` is the worst case of `g/1` as well, otherwise the comparison is meaningless. For instance, it may be possible that the best case of `f/1` is the worst case of `g/1`. The same applies to best cases. Here, we have the exact delays of `rm_fst/2` and `rm_fst_tf/2`. We simply deduce that

$$\mathcal{D}^{\mathsf{rm\_fst}}_{0,-1} = \mathcal{D}^{\mathsf{rm\_fst\_tf}}_{0,-1}, \qquad \mathcal{D}^{\mathsf{rm\_fst}}_{n,k} < \mathcal{D}^{\mathsf{rm\_fst\_tf}}_{n,k},\ \text{ for } n > 0.$$

Therefore, $\mathcal{D}^{\mathsf{rm\_fst}}_{n,k} \leqslant \mathcal{D}^{\mathsf{rm\_fst\_tf}}_{n,k}$, for all $n \geqslant 0$. More precisely, we can study the asymptotic behaviour of

$$\mathcal{D}^{\texttt{rm\_fst\_tf}}_{n,k}/\mathcal{D}^{\texttt{rm\_fst}}_{n,k} = (2k+3)/(k+1) \sim 2, \text{ as } k \to \infty.$$

In other words: $\mathcal{D}^{\texttt{rm\_fst\_tf}}_{n,k} \sim 2 \cdot \mathcal{D}^{\texttt{rm\_fst}}_{n,k}$, as $k \to \infty$ and $k < n$. Asymptotically, `rm_fst/2` is thus twice as fast as `rm_fst_tf/2`, although it needs to store in the call stack n items, whilst `rm_fst_tf/2`, being in tail form, uses a small constant amount of call stack—just one call, actually.

Assuming now that some item I is in the list L, what is the average delay of `rm_fst_tf(`I`,`L`)`? We already expect this to be the arithmetic mean of the delays for all positions. Let us denote $\mathcal{A}^{\texttt{rm\_fst\_tf}}_{n}$ this quantity, which is not an integer in general. Formally, it is defined as

$$\mathcal{A}^{\texttt{rm\_fst\_tf}}_{n} = \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{D}^{\texttt{rm\_fst\_tf}}_{n,k} = \frac{1}{n}\sum_{k=0}^{n-1}(2k+3) = n+2.$$

When the item is present in the list, the worst case happens when it is the last, that is, $k = n-1$, so the delay is then

$$\mathcal{W}^{\texttt{rm\_fst\_tf}}_{n} = \mathcal{D}^{\texttt{rm\_fst\_tf}}_{n,n-1} = 2(n-1)+3 = 2n+1.$$

Therefore, $\mathcal{A}^{\texttt{rm\_fst\_tf}}_{n} \sim \frac{1}{2}\mathcal{W}^{\texttt{rm\_fst\_tf}}_{n}$, as $n \to \infty$. This is the same situation as with $\mathcal{A}^{\texttt{rm\_fst}}_{n}$.

Alternate definition. Let us envisage an alternative definition of `rm_fst/2`:

$$
\begin{array}{lcl}
\texttt{rm\_fst\_alt(I,L)} & \xrightarrow{\alpha} & \texttt{rm\_fst\_\_(I,L,[]).}\\
\texttt{rm\_fst\_\_(\_,\ \ \ [],A)} & \xrightarrow{\beta} & \mathbf{\texttt{A}}\texttt{;}\\
\texttt{rm\_fst\_\_(I,[I|L],A)} & \xrightarrow{\gamma} & \mathbf{\texttt{join(A,L)}}\texttt{;}\\
\texttt{rm\_fst\_\_(I,[J|L],A)} & \xrightarrow{\delta} & \texttt{rm\_fst\_\_(I,L,}\mathbf{\texttt{join(A,[J])}}\texttt{).}
\end{array}
$$

The expressions in bold in clauses β, γ and δ differ from the definition of `rm_fst/3`. The rationale behind its design is to consider the accumulator `A` as a prefix of the original list, not as a reversed prefix as we did before. That is why, instead of pushing `J` on `A` in clause δ, `join/2` is called so that `[J]` is appended *at the end* of `A`. Consequently, the result is `A`, instead of `rev(A)`, in clause β and `join(A,L)`, instead of `rev_join(A,L)`, in clause γ. This solution is appealing on two grounds. Firstly, the order of the items is not reversed, which makes it easier to bear them in mind. Secondly, in the worst case, we avoid reversing two times the items as we did in `rm_fst/3`: the items of the original input were reversed into the accumulator and then the accumulator is reversed to make the final result. Let us compute the delays $\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,k} := [\![\texttt{rm\_fst\_alt(}I\texttt{,}L\texttt{)}]\!]$, where n is the length of L and k is the location where I occurs in L

($k = n$ if absent), and $\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m} := [\![\texttt{rm\_fst(}I\texttt{,}P\texttt{,}Q\texttt{)}]\!]$, where n is the length of P and m the length of Q. There are three cases:

- L is empty;
- L is not empty and I is absent;
- L is not empty and I occurs at position k, counting the head of L as position 0.

In the first case, the execution trace is $\alpha\beta$: the delay is $|\alpha\beta| = 2$. Let us now suppose that I is absent from L, that is, $k = n$.

- clause α is used once;
- clause δ is used n times;
- clause β is used once;
- clause γ is unused .

This execution trace is $\alpha\delta^n\beta$, which must be completed with the call `join(A,L)`. We know that $\mathcal{D}^{\texttt{join}}_q = q+1$, where q is the length of `A`, from equation (3.2) on page 66. Assuming that `L` contains p items, all this translates into the following recurrent equations, where $p, q \geqslant 0$.

$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{p,n} \overset{\alpha}{=} 1 + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{p,0}, \qquad \mathcal{D}^{\texttt{rm\_fst\_\_}}_{0,q} \overset{\beta}{=} 1,$$
$$\mathcal{D}^{\texttt{rm\_fst\_\_}}_{p+1,q} \overset{\delta}{=} 1 + \mathcal{D}^{\texttt{join}}_{q} + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{p,q+1} = 1 + (q+1) + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{p,q+1}.$$

Note that we only used the information about whether a clause is used or not. Now we are going to use the fact that clause δ is used n times. Indeed, these recurrence equations should remind us of equations (3.5) of `rev_join/2` on page 71, albeit the slight difference has important consequences. We can use the same technique, called *telescoping*, on the equation drawn from clause δ, n times:

$$\begin{array}{rl}
 & \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m}} - \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-1,m+1} \overset{\delta}{=} 2 + (m+0) = m+2 \\
+ & \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-1,m+1} - \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-2,m+2} \overset{\delta}{=} 2 + (m+1) = m+3 \\
+ & \qquad\qquad\qquad\vdots \\
+ & \mathcal{D}^{\texttt{rm\_fst\_\_}}_{1,m+(n-1)} - \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{0,m+n}} \overset{\delta}{=} 2 + (m+(n-1)) = m+(n+1) \\
\hline
\Rightarrow & \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m}} - \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{0,m+n}} = nm + \sum_{p=2}^{n+1} p = nm + \sum_{p=1}^{n}(1+p) \\
 & \qquad = nm + \left(n + \sum_{p=1}^{n} p\right)
\end{array}$$

$$\begin{aligned}
\Leftrightarrow \quad & \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m} - 1 = nm + n + n(n+1)/2 \\
\Leftrightarrow \quad & 2 \cdot \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m} = 2 + 2nm + 2n + n^2 + n \\
& = 2nm + n^2 + 3n + 2, \text{ with } n > 0.
\end{aligned}$$

Since that replacing n by 0 in the last formula leads to the expected $\mathcal{D}^{\texttt{rm\_fst\_\_}}_{0,m} \overset{\beta}{=} 1$, we can gather all the cases into just one formula:

$$2 \cdot \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m} = 2nm + n^2 + 3n + 2, \text{ where} n, m \geqslant 0.$$

What we want finally is $\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,n}$ for $n \geqslant 0$:

$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,n} \overset{\alpha}{=} 1 + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,0} = 1 + (n^2 + 3n + 2)/2 = (n^2 + 3n + 4)/2.$$
$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,n} \sim \tfrac{1}{2}n^2, \text{ as } n \to \infty.$$

The asymptotic delay of `rm_fst_alt/2` is therefore quadratic. Now let us suppose that I does occur in L at position k, counting the head of the list as being at position 0. Then $0 \leqslant k < n$ and

- clause α is used once;
- clause δ is used k times;
- clause γ is used once;
- clause β is unused.

The execution trace is $\alpha\delta^k\gamma$, to which must be added the delays of the calls `join(A,L)` and of the repeated calls `join(A,[J])`. We know that $\mathcal{D}^{\texttt{join}}_q = q + 1$, where q is the length of `A`. In clause γ, this argument is `A` and we know that it contains the first k items of L, because clause δ was used k times. Assuming that `L` contains p items, all this translates into the following recurrent equations, where $p, q \geqslant 0$ and k is fixed.

$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{p,k} \overset{\alpha}{=} 1 + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{p,0}, \qquad \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-k,q} \overset{\gamma}{=} 1 + \mathcal{D}^{\texttt{join}}_q = 2 + q,$$
$$\mathcal{D}^{\texttt{rm\_fst\_\_}}_{p+1,q} \overset{\delta}{=} 1 + \mathcal{D}^{\texttt{join}}_q + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{p,q+1} = 2 + q + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{p,q+1}.$$

Note that we only used the information about whether a clause is used or not. Now we are going to use the fact that clause δ is used k times, by telescoping the equation drawn from clause δ, k times, with $0 \leqslant k < n$:

$$\begin{aligned}
& \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m}} - \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-1,m+1} \overset{\delta}{=} 2 + (m+0) = m + 2 \\
+ \quad & \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-1,m+1} - \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-2,m+2} \overset{\delta}{=} 2 + (m+1) = m + 3 \\
+ \quad & \vdots \\
+ \quad & \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-k+1,m+k-1} - \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-k,m+k}} \overset{\delta}{=} 2 + (m + (k-1)) = m + (k+1)
\end{aligned}$$

$$
\begin{aligned}
\Rightarrow \qquad \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m}} - \boxed{\mathcal{D}^{\texttt{rm\_fst\_\_}}_{n-k,m+k}} &= km + \sum_{p=2}^{k+1} p = km + \sum_{p=1}^{k}(1+p)\\
&= km + \left(n + \sum_{p=1}^{k} p\right)\\
\Leftrightarrow \qquad \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m} - (2 + (m+k)) &= km + k + k(k+1)/2, \text{ by } (\overset{\gamma}{=}),\\
\Leftrightarrow \qquad 2 \cdot \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,m} &= k^2 + (2m+5)k + 2m + 4.
\end{aligned}
$$

What we want finally is $\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,k}$:

$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,k} \overset{\alpha}{=} 1 + \mathcal{D}^{\texttt{rm\_fst\_\_}}_{n,0} = 1 + (k^2 + 5k + 4)/2 = (k+2)(k+3)/2.$$

$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,k} \sim \frac{1}{2}k^2, \text{ as } k \to \infty \text{ and } k < n.$$

We can actually extend this asymptotic expression to the case $k = n$, as the previous case analysis proved, so

$$\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,k} \sim \tfrac{1}{2}k^2, \text{ as } k \to \infty \text{ and } k \leqslant n.$$

In the worst case, $k = n - 1$, that is, the item I is at the end of the list L, so the delay is then $(n+1)(n+2)/2 = (n^2 + 3n + 2)/2$. Notice how this is exactly 1 less than the delay when I is missing in L, that is, $\mathcal{D}^{\texttt{rm\_fst\_alt}}_{n,n} = (n^2 + 3n + 4)/2$. Therefore, the worst case happens when I is missing from L. There is no best case when I is absent from L; when present, it is $k = 0$, so the delay is $(0+2)(0+3)/2 = 3$. Thus the best case is when I is the head of L. If the list is empty, that is, $n = 0$, then there is only one case, which is thus both the best and the worst, and the delay is $(0^2 + 3 \cdot 0 + 4)/2 = 2$. For $n > 0$,

$$\mathcal{B}^{\texttt{rm\_fst\_alt}}_0 = 2,\ \mathcal{B}^{\texttt{rm\_fst\_alt}}_n = 3,\ \mathcal{W}^{\texttt{rm\_fst\_alt}}_n = \frac{n^2 + 3n + 4}{2} \sim \frac{1}{2}n^2.$$

This proves that the delay of `rm_fst_alt/2` in the worst case is asymptotically quadratic, as opposed to the linear behaviour of `rm_fst/2`. Thus, despite the items being reversed twice in the latter, the delay is much lower. Just like with `srev/1`, on page 53, the root of the problem here is that `join/2` is moving the same items a number of times which is not constant, actually proportional to their position in the input list, and which, once compounded, yields a quadratic term. Instead, when pushing the items to the accumulator and from it to the result, we have moved each item exactly twice, which is a (small) constant number of times, hence the linear term.

Assuming that the item I is in the list L, what is the expected delay of `rm_fst_alt(`I`,`L`)`? This is the arithmetic mean of the delays for all positions. Let us denote $\mathcal{A}_n^{\texttt{rm\_fst\_alt}}$ this quantity, which is not an integer in general. Mathematically, it is defined as follows:

$$\begin{aligned}\mathcal{A}_n^{\texttt{rm\_fst\_alt}} &:= \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{D}_{n,k}^{\texttt{rm\_fst\_alt}} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{(k+2)(k+3)}{2}\\ &= \frac{1}{2n}\sum_{k=0}^{n-1}(k^2+5k+6) = \frac{1}{2n}\left(\sum_{k=0}^{n-1}k^2 + 5\sum_{k=0}^{n-1}k + 6n\right)\\ &= \frac{1}{2n}\left(\sum_{k=0}^{n-1}k^2 + 5\cdot\frac{n(n-1)}{2} + 6n\right).\end{aligned}$$

We need to find a closed expression of the sum of the first n squares. We can use the telescoping technique as shown on page 89, but on the series $(n^3)_{n>0}$. We start with the equality $(n+1)^3 = n^3+3n^2+3n+1$, hence $(n+1)^3 - n^3 = 3n^2+3n+1$. Then we can sum these differences:

$$\begin{array}{rrcl} & (1+1)^3 - \boxed{1^3} &=& 3\cdot 1^2 + 3\cdot 1 + 1\\ + & (2+1)^3 - 2^3 &=& 3\cdot 2^2 + 3\cdot 2 + 1\\ + & &\vdots& \\ + & \boxed{(n+1)^3} - n^3 &=& 3n^2 + 3n + 1\\ \hline \Rightarrow & \boxed{(n+1)^3} - \boxed{1^3} &=& 3\sum_{k=1}^{n}k^2 + 3\sum_{k=1}^{n}k + n\\ & n^3+3n^2+3n &=& 3\sum_{k=1}^{n}k^2 + 3\cdot\frac{n(n+1)}{2} + n\\ \Leftrightarrow & \sum_{k=1}^{n}k^2 &=& \frac{n(n+1)(2n+1)}{6}.\end{array}$$

We can now resume or calculation of $\mathcal{A}_n^{\texttt{rm\_fst\_alt}}$:

$$\begin{aligned}\mathcal{A}_n^{\texttt{rm\_fst\_alt}} &= \frac{1}{2n}\left(\frac{(n-1)n(2n-1)}{6} + 5\cdot\frac{n(n-1)}{2} + 6n\right)\\ &= \frac{1}{6}n^2 + n + \frac{11}{6} \sim \frac{1}{6}n^2,\ \text{as } n\to\infty.\end{aligned}$$

We found earlier that the worst case when the item is present is when it is located at the end of the list and the delay is then

$$\mathcal{W}_n^{\texttt{rm\_fst\_alt}} = \tfrac{1}{2}n^2 + \tfrac{3}{2}n + 1.$$

Therefore, $\mathcal{A}_n^{\texttt{rm\_fst\_alt}} \sim \frac{1}{3}\mathcal{W}_n^{\texttt{rm\_fst\_alt}}$, as $n \to \infty$. That is to say, asymptotically, the average delay is about 33% the worst delay. In other words, the worst does not happen very often.

Last occurrence. Let us consider the definition of a function `rm_-lst/2` such that the call `rm_lst(`I`,`L`)` is rewritten to L if I does not belong to the list L, otherwise it is rewritten to a list identical to L but without the last occurrence of I. This is our *specification*. For instance, we expect the following rewrites:

$$\begin{aligned}
\texttt{rm\_lst(3,[])} &\twoheadrightarrow \texttt{[];}\\
\texttt{rm\_lst([],[])} &\twoheadrightarrow \texttt{[];}\\
\texttt{rm\_lst([5,2],[3,[]])} &\twoheadrightarrow \texttt{[3,[]];}\\
\texttt{rm\_lst(4,[[],[1,2],4,[],4])} &\twoheadrightarrow \texttt{[[],[1,2],4,[]];}\\
\texttt{rm\_lst([],[4,[1,2],[],[],4])} &\twoheadrightarrow \texttt{[4,[1,2],[],4].}
\end{aligned}$$

Let us go for a direct way to the solution. First of all, does this problem reminds us of another? Filtering out the first occurrence seems close. How then could we reuse `rm_fst/2` to solve our current problem? The key is to see that, provided that the input list is first reversed, applying `rm_fst/2` will filter out the item which was the last occurrence in the original list, since the last occurrence becomes the first due to the reversing. The last step thus consists in reversing the result so the order of the items is restored.

```
rm_lst(I,L) -> rev(rm_fst(I,rev(L))).
```

What is the delay of the calls to this function? Since we compose three functions, the delays are simply added. Let us suppose that the length of the input list L is n. Let us compute the delay $\mathcal{D}_{n,k}^{\texttt{rm\_lst}} := [\![\texttt{rm\_lst(}I\texttt{,}L\texttt{)}]\!]$, where n is the length of L and k is the location where I occurs in L, $k = n$ if absent. Since the first call is `rev(L)`, the definition of `rm_lst/2` leads to the equation

$$\mathcal{D}_{n,k}^{\texttt{rm\_lst}} = 1 + \mathcal{D}_{?}^{\texttt{rev}} + \mathcal{D}_{?,?}^{\texttt{rm\_fst}} + \mathcal{D}_{n}^{\texttt{rev}}.$$

We need to determine the length of the arguments to the composed functions. Because the length of `rev(`L`)` is the same as the length of L, we know that the list argument of `rm_fst/2` has length n, so we have

$$\mathcal{D}_{n,k}^{\texttt{rm\_lst}} = 1 + \mathcal{D}_{?}^{\texttt{rev}} + \mathcal{D}_{n,?}^{\texttt{rm\_fst}} + \mathcal{D}_{n}^{\texttt{rev}}.$$

Now we need to consider different cases. Indeed, depending on the presence or absence of I in L, the length of `rm_fst(`I`,rev(`L`))` is different: if present, it is $n - 1$, otherwise it is n. But there is another case to consider first: the input list may be empty. In this case, $n = k = 0$, but this is not interpreted as I being the head of L. We have

$$\mathcal{D}_{0,0}^{\texttt{rm\_lst}} = 1 + 2 + 1 + 2 = 6.$$

Next, let us assume that I is absent, that is, $k = n$. We found on page 87:

$$\mathcal{D}_{n,k}^{\texttt{rm\_fst}} = k + 1, \text{ with } 0 \leqslant k \leqslant n,$$

where k is the position at which I occurs last in P in the call `rm_-fst(`I`,`P`)`, with the convention $k = n$ if it does not. Thus, $\mathcal{D}_{n,n}^{\texttt{rm\_fst}} = n+1$ and

$$\begin{aligned}\mathcal{D}_{n,n}^{\texttt{rm\_lst}} &= 1 + \mathcal{D}_{n}^{\texttt{rev}} + \mathcal{D}_{n,n}^{\texttt{rm\_fst}} + \mathcal{D}_{n}^{\texttt{rev}}\\ &= 1 + (n+2) + (n+1) + (n+2) = 3n + 6, \text{ with } n > 0.\end{aligned}$$

Because we just found that $\mathcal{D}_{0,0}^{\texttt{rm\_lst}} = 6$, we can extend this formula to the case $n = 0$, that is, the empty list:

$$\mathcal{D}_{n,n}^{\texttt{rm\_lst}} = 3n + 6, \text{ with } n \geqslant 0.$$

Let us suppose now that I occurs last in L at position k. Then the length of `rm_fst(`I`,rev(`L`))` is $n - 1$. Furthermore, I occurs in `rev(`L`)` at position $n-k-1$, so $\mathcal{D}_{n,n-k-1}^{\texttt{rm\_fst}} = (n-k-1)+1 = n-k$. We deduce, for $0 \leqslant k < n$,

$$\begin{aligned}\mathcal{D}_{n,k}^{\texttt{rm\_lst}} &= 1 + \mathcal{D}_{n-1}^{\texttt{rev}} + \mathcal{D}_{n,n-k-1}^{\texttt{rm\_fst}} + \mathcal{D}_{n}^{\texttt{rev}}\\ &= 1 + ((n-1)+2) + (n-k) + (n+2) = 3n - k + 4. \qquad (4.9)\end{aligned}$$

Let us note $\mathcal{B}_{n}^{\texttt{rm\_lst}}$ and $\mathcal{W}_{n}^{\texttt{rm\_lst}}$ the delays, respectively, in the best and worst cases. By definition, they are, respectively, the minimum and the maximum delay, when n is fixed and k varies, that is

$$\begin{aligned}\mathcal{B}_{n}^{\texttt{rm\_lst}} &:= \min_{0\leqslant k\leqslant n} \mathcal{D}_{n,k}^{\texttt{rm\_lst}} = \min_{0\leqslant k< n} \{3n+6, 3n-k+4\}\\ &= 3n - (n-1) + 4 = 2n + 5 \sim 2n, \text{ as } n \to \infty;\\ \mathcal{W}_{n}^{\texttt{rm\_lst}} &:= \max_{0\leqslant k\leqslant n} \mathcal{D}_{n,k}^{\texttt{rm\_lst}} = \max_{0\leqslant k< n} \{3n+6, 3n-k+4\}\\ &= 3n + 6 \sim 3n, \text{ as } n \to \infty.\end{aligned}$$

The best case is thus when $k = n - 1$, that is, when the item is the last in the list, and the worst case happens when the item is missing. Let us denote $\mathcal{A}_{n}^{\texttt{rm\_lst}}$ the average delay, which is the arithmetic mean of the delays for all positions. Mathematically, it is defined as follows:

$$\begin{aligned}\mathcal{A}_{n}^{\texttt{rm\_lst}} &= \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{D}_{n,k}^{\texttt{rm\_lst}} = \frac{1}{n}\sum_{k=0}^{n-1}(3n-k+4) = 3n + 4 - \frac{1}{n}\sum_{k=0}^{n-1}k\\ &= 3n + 4 - \frac{n-1}{2} = \frac{5}{2}n + \frac{9}{2} \sim \frac{5}{2}n, \text{ as } n \to \infty. \qquad (4.10)\end{aligned}$$

When the item is present, the worst case is when it is the head:

$$\mathcal{W}_n^{\texttt{rm\_lst}} = \mathcal{D}_{n,0}^{\texttt{rm\_lst}} = 3n + 4 \sim 3n, \text{ as } n \to \infty.$$

This implies $\mathcal{A}_n^{\texttt{rm\_lst}} \sim \frac{5}{6}\mathcal{W}_n^{\texttt{rm\_lst}}$, as $n \to \infty$, which means that, unfortunately, the average delay is asymptotically 83.3% of the worst delay. Contrast this with the average delay of `rm_fst/2`, equation (4.8) on page 88, which is only 50% of the worst delay.

Adding a sequential search. We found that the worst case of `rm_lst/2` happens when the item is missing in the list of length n and the delay is then $3n + 6$. This means that three complete traversals of the lists are made, whilst only one would be enough to realise that the item is indeed absent. Similarly, the best case is when the item occurs in the last position in the list, in which case the delay is $2n + 5$, that is, two complete traversals are performed, despite only one would be necessary. This suggests that there may be some room for improvement. Perhaps the first try consists in keeping the previous algorithm while adding some shortcuts. For instance, we can delay the composition of `rev/1` and `rm_fst/2` until we actually find an occurrence of the item; otherwise, we keep going until the end of the list:

```
rm_lst1(_,   []) -α-> [];
rm_lst1(I,[I|L]) -β-> rev(rm_fst(I,rev([I|L])));
rm_lst1(I,[J|L]) -γ-> [J|rm_lst1(I,L)].
```

The part of the definition not in bold is the typical *sequential search* we have seen at play in `rm_fst/2`: all the items of a list are visited in turn until one of interest is found. Importantly, the visited items are not discarded, but kept, with the same relative order, in the control context of the last clause. The body in bold is the action we want to perform in case we find the item we sought. What is the delay of `rm_lst1/2`? As usual in sequential searches, two cases are a priori relevant: either the item is present or not.

Let us assume that the item occurs first at position f in the list and last at position k, counting the head as position 0. Clause γ is then used f times and clause β once. Then `rev(rm_fst(I,rev([I|L])))` is rewritten into a value. The delay of reversing `[I|L]` is $(n - f) + 2$. The last occurrence of `I` in `[I|L]` is $k - f$, so the first occurrence in `rev([I|L])` is $((n - 1) - f) - (k - f) = n - k - 1$. We can check these expressions on a small example in figure 18 on page 100. We used `e`, `a`, `b` etc. for the contents of the list, instead of numbers, in order to avoid confusion with the indexes. The item whose last occurrence is k is `b`. We already know that $\mathcal{D}_{n,k}^{\texttt{rm\_fst}} = k + 1$. Let $\mathcal{D}_{n,f,k}^{\texttt{rm\_lst1}}$ be the delay when the item occurs first in position f and last in position k in a list

		f			k		n
0	1	2	3	4	5	6	7
e	a	**b**	d	c	**b**	r	
6	5	4	3	2	1	0	
		0	1	2	3	4	
		4	3	2	1	0	

FIGURE 18: Filtering out the last b in eabdcbr

of length n. We have

$$\begin{aligned}\mathcal{D}^{\text{rm\_lst1}}_{n,f,k} &= f + 1 + \mathcal{D}^{\text{rev}}_{n-f} + \mathcal{D}^{\text{rm\_fst}}_{n-f,n-k-1} + \mathcal{D}^{\text{rev}}_{n-f-1}\\ &= f + 1 + ((n-f)+2) + ((n-k-1)+1)\\ &\qquad + ((n-f-1)+2) = 3n - f - k + 4, \text{ with } n > 0.\end{aligned}$$

We found earlier that the delay of `rm_lst/2`, when the item was occurring last at position k in a list of length n was $3n - k + 4$. Thus, we have in this case $\mathcal{D}^{\text{rm\_lst1}}_{n,f,k} = \mathcal{D}^{\text{rm\_lst}}_{n,k} - f$. This was expected because the reduction in delay due to f comes from the wrapping of a sequential search for the item around `rm_lst/2`.

Let us suppose now that the item does not occur in the list. Then clause β is unused and we have a sequential search ending in a failure, that is, $\mathcal{D}^{\text{rm\_lst1}}_{n,n,n} = n + 1$. (Using the convention that an occurrence at position n means "absent.") If we suppose now that the item does occur in the list, first at position f and last at position k, then, by definition, the best case happens when $\mathcal{D}^{\text{rm\_lst1}}_{n,f,k}$ is minimum and the worst case when it is maximum:

$$\begin{aligned}\mathcal{B}^{\text{rm\_lst1}}_{n} &:= \min_{0 \leqslant f \leqslant k \leqslant n} \mathcal{D}^{\text{rm\_lst1}}_{n,f,k} = \min_{0 \leqslant f \leqslant k < n} \{n+1, 3n - f - k + 4\}\\ &= \min\{n+1, 3n - (n-1) - (n-1) + 4\} = n + 1.\\ \mathcal{W}^{\text{rm\_lst1}}_{n} &:= \max_{0 \leqslant f \leqslant k \leqslant n} \mathcal{D}^{\text{rm\_lst1}}_{n,f,k} = \max_{0 \leqslant f \leqslant k < n} \{n+1, 3n - f - k + 4\}\\ &= \max\{n+1, 3n - 0 - 0 + 4\} = 3n + 4.\end{aligned}$$

As a conclusion, the best case of `rm_lst1/2` happens when the item is missing and the worst case when the item occurs *uniquely* as the head of the list ($f = k = 0$). Notice that it is not possible to compare these results with the delays of `rm_lst/2` because the best and worst cases differ (the worst case of `rm_lst/2` is the best case of `rm_lst1/2`) and

$$\mathcal{B}^{\text{rm\_lst1}}_{n} < \mathcal{B}^{\text{rm\_lst}}_{n} < \mathcal{W}^{\text{rm\_lst1}}_{n} < \mathcal{W}^{\text{rm\_lst}}_{n}.$$

(It would have been possible to compare `rm_lst/2` and `rm_lst1/2` in an absolute sense had the worst delay of one been smaller than the best delay of the other.) Perhaps an analysis of the average delay would enable a comparison.

Assuming that the item occurs at least once in the list, what is the average delay of `rm_lst1/2`? The first occurrence position, f, ranges from 0 to $n-1$, whilst the last occurrence position k ranges from f to $n-1$. The pairs (f,k) can be counted as follows: $(0,0)\ldots(0,n-1)$, then $(1,1)\ldots(1,n-1)$, etc. until $(n-1,n-1)$. Thus the number of pairs with $f=0$ is n, the number of pairs with $f=1$ is $n-1$ etc. up to 1 when $f=n-1$. This means that the total number of pairs (f,k) is $1+2+\cdots+n = n(n+1)/2$. We can now lay out the definition

$$
\begin{aligned}
\mathcal{A}_n^{\texttt{rm\_lst1}} &:= \frac{2}{n(n+1)}\sum_{f=0}^{n-1}\sum_{k=f}^{n-1}\mathcal{D}_{n,f,k}^{\texttt{rm\_lst}} = \frac{2}{n(n+1)}\sum_{f=0}^{n-1}\sum_{k=f}^{n-1}(3n-f-k+4)\\
&= 3n+4-\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\sum_{k=f}^{n-1}(f+k)\\
&= 3n+4-\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(f(n-f)+\sum_{k=f}^{n-1}k\right)\\
&= 3n+4-\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(nf-f^2+\frac{n(n-1)}{2}-\frac{f(f-1)}{2}\right)\\
&= 3n+4-\frac{2}{n(n+1)}\cdot\frac{n^2(n-1)}{2}\\
&\qquad+\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(\frac{3}{2}f^2-\left(\frac{1}{2}+n\right)f\right)\\
&= 3n+4-\frac{n(n-1)}{n+1}+\frac{3}{n(n+1)}\sum_{f=0}^{n-1}f^2-\frac{2n+1}{n(n+1)}\sum_{f=0}^{n-1}f\\
&= 3n+4-\frac{n(n-1)}{n+1}+\frac{3}{n(n+1)}\cdot\frac{(n-1)n(2n-1)}{6}\\
&\qquad-\frac{2n+1}{n(n+1)}\cdot\frac{n(n-1)}{2}\\
&= 3n+4-\frac{n(n-1)}{n+1}+\frac{(n-1)(2n-1)}{2(n+1)}-\frac{(n-1)(2n+1)}{2(n+1)}\\
&= 3n+4+\frac{(n-1)(-2n+2n-1-2n-1)}{2(n+1)} = 2n+5.
\end{aligned}
$$

We found earlier that $\mathcal{A}_n^{\texttt{rm\_lst}} = \frac{5}{2}n+\frac{9}{2}$, therefore $\mathcal{A}_n^{\texttt{rm\_lst1}} < \mathcal{A}_n^{\texttt{rm\_lst}}$, for $n>1$. In other words, `rm_lst1/2` is in average faster than `rm_lst/2`.

Finally, we remark that $\mathcal{A}_n^{\texttt{rm\_lst1}} \sim \frac{2}{3}\mathcal{W}_n^{\texttt{rm\_lst1}}$, as $n \rightarrow \infty$, that is, the average delay of `rm_lst1/2` is 66% of the worst delay.

Tail form. For the sake of further training, let us transform the definition of `rm_lst/2` into tail form. The starting point consists in copying the original definition and rename it `rm_lst_tf/2`:

```
rm_lst_tf(I,L) -> rev(rm_fst(I,rev(L))).
```

The first call to be performed is `rev(L)`. The first thing to check is whether `rev/1` is in tail form or not. It is, because we found the following, on page 60:

```
rev(L)            -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Secondly, we locate the control context of the innermost call, that is, `rev(L)`: it is `rev(rm_fst(I,␣))`. It is handy to visualise on the code the variables it contains by framing them, while the first call is in bold:

```
rm_lst_tf(I,L) -> rev(rm_fst(I,rev(L))).
```

Now let us add a list accumulator to `rev/1`, whose purpose is to hold the variables present in the control context of the innermost call. Here, there is only one variable, `I`:

```
rm_lst_tf(I,L)    -> rev(rm_fst(I,rev(L,[I]))).
rev(L,A)          -> rev_join(L,[]).          % A still unused
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Let us follow the clauses that rewrite the innermost call, `rev(L,[I])` and let us inscribe within a box the first body which is not a recursive call:

```
rm_lst_tf(I,L)    -> rev(rm_fst(I,rev(L,[I]))).
rev(L,A)          -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Since the definition of `rev/2` is made of a unique clause, we can move the control context of `rev(L,[I])` to apply to it, for which we need to use the variable `I` in the accumulator:

```
rm_lst_tf(I,L)    -> rev(L,[I]).
rev(L,[I])        -> rev(rm_fst(I,rev_join(L,[]))).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

We must start again the same operations on the new body where the innermost call is `rev_join(L,[])` and whose control context is `rev(rm_-fst(I,␣))`. We must check whether the call to perform first is in tail form or not. The definition of `rev_join/2` is in tail form, so the next step is to add an accumulator `A` to `rev_join/2`, initialised with the variables in the control context (here, only `I`). This accumulator is passed, unchanged to any recursive calls:

```
rm_lst_tf(I,L)      -> rev(L,[I]).
rev(L,[I])          -> rev(rm_fst(I,rev_join(L,[],[I]))).
rev_join(   [],Q,A) -> Q;                          % A yet unused
rev_join([I|P],Q,A) -> rev_join(P,[I|Q],A).          % Unchanged
```

Then we repeat the same steps, that is, we identify the first bodies used to rewrite the innermost call, that is, `rev_join(L,[],[I])`, which are not recursive (this is why it is paramount that `rev_join/2` is already in tail form: its bodies are either a call without control context or a value or an arithmetic expression):

```
rm_lst_tf(I,L)      -> rev(L,[I]).
rev(L,[I])          -> rev(rm_fst(I,rev_join(L,[],[I]))).
rev_join(   [],Q,A) -> Q;
rev_join([I|P],Q,A) -> rev_join(P,[I|Q],A).
```

Next, we move the control context `rev(rm_fst_tf(I,␣))` to apply to the framed bodies (here, only `Q`), making sure that the variable `I` is bound with the help of the accumulator `A`:

```
rm_lst_tf(I,L)        -> rev(L,[I]).
rev(L,[I])            -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rev(rm_fst(I,Q));
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
```

The good news is that the only body which is not in tail form now is made of only two calls instead of three originally. Now the innermost call is `rm_fst(I,Q)`, so we must check first whether `rm_fst/2` is in tail form. It is not, as we found on page 80:

```
rm_fst(_,   []) -> [];
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].           % Not in tail form
```

Therefore, we must transform now the definition of `rm_fst/2` into an equivalent one in tail form. If we don’t, this is what happens: the control context `rev(␣)` is moved so it applies to the bodies of `rm_fst/2` which contain no calls. We have

```
rm_lst_tf(I,L)        -> rev(L,[I]).
```

```
rev(L,[I])            -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rm_fst(I,Q);
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
rm_fst(_,   [])       -> rev([]);                       % Wrong
rm_fst(I,[I|L])       -> rev(L);                        % Wrong
rm_fst(I,[J|L])       -> [J|rm_fst(I,L)].
```

This is wrong because the last clause of rm_fst/2 is not in tail form, so when no calls are needed, as in the two first clauses of the original rm_-fst/2, there is still work to do: the control context [J|␣] holds values “waiting” to be pushed. But, what we meant here was to apply rev/1 to the *value* of rm_fst(I,Q), that is, the final result, when nothing else is left to be computed by rm_fst/2—hence the error. The version of rm_fst/2 in tail form was already encountered on page 85 and named rm_fst_tf/2:

```
rm_fst_tf(I,L)    -> rm_fst(I,L,[]).
rm_fst(_,   [],A) -> rev(A);
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst(I,L,[J|A]).
```

So we now simply change the call rm_fst(I,Q) in the definition above of rev_join/3 into the call rm_fst_tf(I,Q) and resume our process:

```
rm_lst_tf(I,L)        -> rev(L,[I]).
rev(L,[I])            -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rev(rm_fst_tf(I,Q));         % Renaming
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
```

Because rm_fst_tf/2 is defined by a single clause, we can replace the call rm_fst_tf(I,Q) by its definition:

```
rm_lst_tf(I,L)        -> rev(L,[I]).
rev(L,[I])            -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rev(rm_fst(I,Q,[]));
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
```

Since rm_fst/3 is also in tail form and since the control context rev(␣) contains no variable, we can repeat the same operation and move the control context so it applies to the non-recursive bodies of rm_fst/3:

```
rm_lst_tf(I,L)        -> rev(L,[I]).
rev(L,[I])            -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rm_fst(I,Q,[]);
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
rm_fst(_,   [],A)     -> rev(rev(A));             % Added context
rm_fst(I,[I|L],A)     -> rev(rev_join(A,L));     % Added context
rm_fst(I,[J|L],A)     -> rm_fst(I,L,[J|A]).
```

The new definition of `rm_fst/3` is not in tail form, but we can trust that composing twice `rev/1` on a list yields a list equal to the original list, so `rev(rev(A))` is equivalent to `A`. Moreover, a little figure on paper would convince us that `rev(rev_join(A,L))` is equivalent to `rev_join(L,A)`, that is, these two expressions are always rewritten to the same value. Finally, taking into account these simplifications, we reach

```
rm_lst_tf(I,L)        -> rev(L,[I]).
rev(L,[I])            -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rm_fst(I,Q,[]);
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
rm_fst(_,   [],A)     -> A;          % Algebraic simplification
rm_fst(I,[I|L],A)     -> rev_join(L,A);                   % Idem
rm_fst(I,[J|L],A)     -> rm_fst(I,L,[J|A]).
```

All these definitions are in tail form now. A quick inspection reveals that `rev/2` is called only once and is defined by means of a single clause, thus, we can replace its only call by its body:

```
rm_lst_tf(I,L)        -> rev_join(L,[],[I]).
rev_join(   [],Q,[I]) -> rm_fst(I,Q,[]);
rev_join([I|P],Q,  A) -> rev_join(P,[I|Q],A).
rm_fst(_,   [],A)     -> A;
rm_fst(I,[I|L],A)     -> rev_join(L,A);
rm_fst(I,[J|L],A)     -> rm_fst(I,L,[J|A]).
```

Note how we actually took care of replacing the call by the body *where the parameters are replaced by their corresponding arguments.* Moreover, consider how, in the first clause of `rev_join/3`, `Q` becomes the input list of the new `rm_fst/3`. There is still room for a tiny improvement: the accumulator of `rev_join/3` named `A` always contains exactly one item: `I`. So why not replace it by `I` itself? Also, in order enhance legibility, we can rename variable `L` in `rm_fst/3` into `Q` and variable `I` in `rev_join/3` into `J`. Finally, we obtain these simpler tail form definitions:

```
rm_lst_tf(I,L)       $\xrightarrow{\alpha}$ rev_join(L,[],I).
rev_join(   [],Q,I)  $\xrightarrow{\beta}$ rm_fst(I,Q,[]);
rev_join([J|P],Q,I)  $\xrightarrow{\gamma}$ rev_join(P,[J|Q],I).
rm_fst(_,   [],A)    $\xrightarrow{\delta}$ A;
rm_fst(I,[I|Q],A)    $\xrightarrow{\epsilon}$ rev_join(Q,A);
rm_fst(I,[J|Q],A)    $\xrightarrow{\zeta}$ rm_fst(I,Q,[J|A]).
```

Delays. Let us compute the delay $\mathcal{D}_{n,k}^{\texttt{rm\_lst\_tf}} := [\![\texttt{rm\_lst\_tf}(I,L)]\!]$, where n is the length of L and k is the greatest location where I occurs in L ($k = n$ if absent). There are three cases to consider:

- L is empty;
- L is not empty and I is absent;
- L is not empty and I occurs at position k, counting the head of L as position 0.

If the list is empty, clause α is used once, clause β is used once and clause δ is used once. In total, the delay is 3. In other words: $\mathcal{D}_{0,0}^{\texttt{rm\_lst\_tf}} = 3$.
If the list L is not empty and I is absent, that is, $k = n > 0$, then

- clause α is used once;
- clause γ is used n times;
- clause β is used once;
- clause ζ is used n times;
- clause δ is used once.

This execution trace is formally $\alpha\gamma^n\beta\zeta^n\delta$. In total, $\mathcal{D}_{n,n}^{\texttt{rm\_lst\_tf}} = 2n+3$, with $n > 0$. We see that this formula is valid even if $n = 0$, so we can merge the two first cases into one: $\mathcal{D}_{n,n}^{\texttt{rm\_lst\_tf}} = 2n+3$, with $n \geqslant 0$. If the list L is not empty and the item I lastly occurs in L at position k, with $0 \leqslant k < n$, assuming that the head has position 0, then

- clause α is used once,
- clause γ is used n times,
- clause β is used once,
- clause ζ is used $n-k$ times,
- clause ϵ is used once.

This is $\alpha\gamma^n\beta\zeta^{n-k}\epsilon$. The delay of the call `rev_join(Q,A)` in the body of clause ϵ must be counted and added. Since `Q` contains k items, equation (3.6) on page 71 states that $\mathcal{D}_k^{\texttt{rev\_join}} = k+1$, so the total is

$$\mathcal{D}_{n,k}^{\texttt{rm\_lst\_tf}} = 1+n+1+(n-k)+1+(k+1) = 2n+4, \text{ with } n > 0.$$

Interestingly, the delays here do not depend on k, that is, the position of the last occurrence of the item. This is not the case for `rm_lst/2`, as equation (4.9) on page 98 is

$$\mathcal{D}_{n,k}^{\texttt{rm\_lst}} = 3n-k+4, \text{ with } 0 \leqslant k \leqslant n.$$

The worst case for `rm_lst_tf/2` happens when the item is present *anywhere* in the list, because $2n+3 < 2n+4$. In any case,

$$\mathcal{D}_{n,k}^{\texttt{rm\_lst\_tf}} \sim 2n, \text{ as } n \to \infty \text{ and } k < n.$$

Note that there are only two possible delays, for any input. Since the position of the item in the list does not impact the delay of `rm_lst_tf/2`, the average delay is the normal delay:

$$\mathcal{A}_n^{\texttt{rm\_lst\_tf}} = \mathcal{D}_{n,k}^{\texttt{rm\_lst\_tf}} = 2n + 4.$$

Equation (4.10) on page 98 was $\mathcal{A}_n^{\texttt{rm\_lst}} = \frac{5}{2}n + \frac{9}{2}$, so it yields now

$$\mathcal{A}_n^{\texttt{rm\_lst\_tf}} \sim \tfrac{4}{5}\mathcal{A}_n^{\texttt{rm\_lst}}, \text{ as } n \rightarrow \infty,$$

which means that the average delay of `rm_lst_tf/2` is 80% of the average delay of `rm_lst/2`—in other words, it is 20% smaller.

As we found above, the worst case of `rm_lst_tf/2` when the item is present does not depend on its position inside the list. This in stark contrast with `rm_lst/2`, whose worst case is when the item is the head of the list. The best case for both functions is when the item is absent from the list. Let us note $\mathcal{B}_n^{\texttt{rm\_lst\_tf}}$ the delay in the best case for an input of size n and $\mathcal{W}_{\overline{n}}^{\text{lst\_tf}}$ the delay in the worst case. We found that, for $n > 0$,

$$\mathcal{B}_n^{\texttt{rm\_lst\_tf}} = 2n + 3 \sim 2n, \quad \mathcal{W}_n^{\texttt{rm\_lst\_tf}} = 2n + 4 \sim 2n, \text{ as } n \rightarrow \infty.$$

We can now precisely compare `rm_lst_tf/2` and `rm_lst/2` in terms of delay. We have, for $n > 0$,

$$\begin{aligned} \mathcal{B}_n^{\texttt{rm\_lst}} &= 2n + 5, & \mathcal{W}_n^{\texttt{rm\_lst}} &= 3n + 6,\\ \mathcal{B}_n^{\texttt{rm\_lst\_tf}} &= 2n + 3, & \mathcal{W}_n^{\texttt{rm\_lst\_tf}} &= 2n + 4. \end{aligned}$$

The keen reader may object now that the worst case of `rm_lst/2` is *not* the same as for `rm_lst_tf/2`, since, for the former, it is defined as the item being the head of the list, whilst for the latter it is when the item occurs *anywhere* in the list. Indeed. To understand why, imagine that the last occurrence of the item is not at the head of the list, whereas it is one possible worst case for `rm_lst_tf/2` but not for `rm_lst/2`. Therefore, `rm_lst/2` might outperform `rm_lst_tf/2` on this particular example. Nevertheless, we can compare both functions globally because the worst delay of `rm_lst_tf/2` is smaller than the best delay of `rm_lst/2`:

$$\mathcal{W}_n^{\texttt{rm\_lst\_tf}} < \mathcal{B}_n^{\texttt{rm\_lst}}.$$

We can also compare directly the two functions on the *intersection* of their worst cases, that is, when the item we look for is the head of the list. In this special case, then a sensible difference is likely, especially for large values of n, as

$$\mathcal{W}_n^{\texttt{rm\_lst\_tf}} \sim \tfrac{2}{3}\mathcal{W}_n^{\texttt{rm\_lst}}, \text{ as } n \rightarrow \infty.$$

We can compare directly $\mathcal{B}_n^{\texttt{rm\_lst}}$ and $\mathcal{B}_n^{\texttt{rm\_lst\_tf}}$, because these delays correspond to the same case, that is, when the item is missing from the non-empty list, and we also find that `rm_lst_tf/2` is barely faster than `rm_lst/2`.

As a conclusion, the version in tail form is always more efficient (between 0% and 33.3%, in average 20%), although we said earlier that it should not be surprising that it is in fact slower, since the main purpose of being in tail form is to save call stack space, not rewrites. The reason why `rm_lst_tf/2` is both faster and uses a small constant amount of call stack is because we did not follow our method until the end and, instead, relied on the following algebraic identities, used from left to right, for a speedup:

$$\begin{aligned}\texttt{rev(rev(}P\texttt{))} &\equiv P,\\ \texttt{rev(rev\_join(}P\texttt{,}Q\texttt{))} &\equiv \texttt{rev\_join(}Q\texttt{,}P\texttt{)},\end{aligned}$$

for all lists P and Q.

We could wonder now: what if we added a sequential search to `rm_-lst_tf/2`? Here is the definition again, as found on the previous page:

```
rm_lst_tf(I,L)       -> rev_join(L,[],I).
rev_join(   [],Q,I) -> rm_fst(I,Q,[]);
rev_join([J|P],Q,I) -> rev_join(P,[J|Q],I).
rm_fst(_,   [],A)   -> A;
rm_fst(I,[I|Q],A)   -> rev_join(Q,A);
rm_fst(I,[J|Q],A)   -> rm_fst(I,Q,[J|A]).
```

We could save time by not reversing `L` by means of `rev_join/3` and instead start with a sequential search for item `I`. If it is found, then *the rest of the list* is worth reversing. This would be

$$\begin{aligned}
&\texttt{rm\_lst2(\_,~~~[])} \xrightarrow{\alpha} \texttt{[];}\\
&\texttt{rm\_lst2(I,[I|L])} \xrightarrow{\beta} \mathbf{\texttt{rm\_fst(I,rev\_join(L,[I]),[]);}}\\
&\texttt{rm\_lst2(I,[J|L])} \xrightarrow{\gamma} \texttt{[J|rm\_lst2(I,L)].}\\
&\texttt{rm\_fst(\_,~~~[],A)} \xrightarrow{\delta} \texttt{A;}\\
&\texttt{rm\_fst(I,[I|Q],A)} \xrightarrow{\epsilon} \texttt{rev\_join(Q,A);}\\
&\texttt{rm\_fst(I,[J|Q],A)} \xrightarrow{\zeta} \texttt{rm\_fst(I,Q,[J|A]).}
\end{aligned}$$

Again, the part of the definition not in bold is the typical scheme of a sequential search, whilst the body in bold is a specific action performed if the first occurrence of the sought item is found. Note that we did not care of preserving the tail form, but only to reuse `rm_fst/2`. What is the delay of `rm_lst_tf/2`? If the item is absent from the list, the delay is that of a failed sequential search in a list of length n, that is, $n + 1$. If the item occurs first at position f, the delay is

$$\mathcal{D}_n^{\texttt{rm\_lst2}} = f + \mathcal{D}_{n-f-1}^{\texttt{rev\_join}} + \mathcal{D}_{n-f}^{\texttt{rm\_fst}}.$$

We already know that $\mathcal{D}_n^{\texttt{rev\_join}} = n+1$, so

$$\mathcal{D}_n^{\texttt{rm\_lst2}} = f + ((n-f-1)+1) + \mathcal{D}_{n-f}^{\texttt{rm\_fst}} = n + \mathcal{D}_{n-f}^{\texttt{rm\_fst}}.$$

The execution trace of `rm_fst/2` in this context is $\zeta^{n-1-k}\epsilon$, followed by the trace of the call `rev_join(Q,A)` in clause ϵ, so

$$\mathcal{D}_n^{\texttt{rm\_fst}} = (n-1-k)+1+\mathcal{D}_{k-f}^{\texttt{rev\_join}} = n-k+((k-f)+1) = n-f+1,$$

where the length of `Q` is $k-f$. Finally, $\mathcal{D}_n^{\texttt{rm\_lst2}} = n+(n-k+1) = 2n-k+1$. Therefore, the best and worst delays *when the item is present* are $\mathcal{B}_n^{\texttt{rm\_lst2}} = 2n-(n-1)+1 = n+2, \mathcal{W}_n^{\texttt{rm\_lst2}} = 2n+1$. Assuming now that the item is absent from the list, the delay is simply that of a failed sequential search, that is, $\mathcal{D}_n^{\texttt{rm\_lst2}} = n+1$. Which allows us to conclude that, in any case, for $n \geqslant 0$,

$$\mathcal{B}_n^{\texttt{rm\_lst2}} = n+1, \qquad \mathcal{W}_n^{\texttt{rm\_lst2}} = 2n+1.$$

The average delay when the item occurs in the list is

$$\begin{aligned}\mathcal{A}_n^{\texttt{rm\_lst2}} &:= \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{D}_k^{\texttt{rm\_lst2}} = \frac{1}{n}\sum_{k=0}^{n-1}(2n-k+1)\\ &= 2n+1-\frac{1}{n}\sum_{k=0}^{n-1}k = 2n+1-\frac{n-1}{2} = \frac{3}{2}n+\frac{3}{2}.\end{aligned}$$

Up to now, the findings can be related in the table of FIGURE 19 on page 109, bearing in mind that two best cases or two worst delays cannot always be compared because the configuration of the input may be different. We can deduce nevertheless that `rm_lst2/2` is always faster than `rm_lst/2` and `rm_lst_tf/2` because $\mathcal{W}_n^{\texttt{rm\_lst2}} < \mathcal{B}_n^{\texttt{rm\_lst}}$ and $\mathcal{W}_n^{\texttt{rm\_lst2}} < \mathcal{B}_n^{\texttt{rm\_lst\_tf}}$. Moreover, two average delays can always be compared (they suppose here that the item does occur in the list), thus `rm_lst2/2` is the fastest of all in average. Finally, we want to insist

	Delay				
	Item				
Function	Present	Absent	Best	Worst	Average
`rm_lst/2`	$3n-k+4$	$3n+6$	$2n+5$	$3n+6$	$\frac{5}{2}n+\frac{9}{2}$
`rm_lst1/2`	$3n-f-k+4$	$n+1$	$n+1$	$3n+4$	$2n+5$
`rm_lst_tf/2`	$2n+4$	$2n+3$	$2n+3$	$2n+4$	$2n+4$
`rm_lst2/2`	$2n-k+1$	$n+1$	$n+1$	$2n+1$	$\frac{3}{2}n+\frac{3}{2}$

FIGURE 19: Delays of different definitions for filtering out an item

again that “average delay” should not be confused with “delay of the average case,” because it might be that there is no average case, that is, a configuration of the input entailing a delay which is the average delay. The average delay is defined as an arithmetic computation the delays corresponding to all possible input configurations, under some equiprobability hypotheses.

Tail form redux. Until now we have seen slightly different ways to convert a given definition into tail form:

1. from `sum/1` to `sum_tf/1` and from `fact/1` to `fact_tf/1`;
2. from `join/2` to `join_tf/2`;
3. from `rm_fst/2` to `rm_fst_tf/2`;
4. from `rm_lst/2` to `rm_lst_tf/2`.

All these transformations rely on an additional parameter but their design seem to differ in the nature of this accumulator: it represents either a partial result or a temporary list. The reason for the dissimilarity actually lies in the use of dedicated improvements for each case, but there is a common scheme for all these transformations into tail form.

Factorial. For instance, if we do not rely on associativity of the arithmetic operators involved in `sum/1` and `fact/1`, that is, addition and multiplication, we then need to keep the integers in a list so that they can be processed in the proper, reversed, order. First, let us recall the original definition of `fact/1`:

```
fact(1)            $\xrightarrow{\alpha}$ 1;
fact(N) when N > 1 $\xrightarrow{\beta}$ N * fact(N-1).
```

and the example on page 9:

```
fact(5) $\xrightarrow{\beta}$ 5 * fact(4)
        $\xrightarrow{\beta}$ 5 * (4 * fact(3))
        $\xrightarrow{\beta}$ 5 * (4 * (3 * fact(2)))
        $\xrightarrow{\beta}$ 5 * (4 * (3 * (2 * fact(1))))
        $\xrightarrow{\alpha}$ 5 * (4 * (3 * (2 * (1))))    = 120.
```

We derived the following definition of `fact_tf/1` in tail form, on page 14:

```
fact_tf(N) when N > 0 -> fact_tf(N-1,N).
fact_tf(0,A)          -> A;
fact_tf(N,A)          -> fact_tf(N-1,A*N).
```

The order in which the multiplications are carried out by `fact_tf/2` is not the same as done by `fact/1`, as it was assumed that multiplication is associative. If we do want not rely on associativity, we need to create and store the integers in a list and then multiply them, for instance, by reusing `mult_tf/1`, defined on page 32. This way the multiplications will be performed exactly in the same order as in `fact/1` because pushing the numbers on the accumulator actually reverses their order:

$$
\begin{array}{ll}
\texttt{fact\_alt\_tf(N) when N > 0} & \xrightarrow{\alpha} \texttt{fact\_alt\_tf(N-1,}\mathbf{\texttt{[N]}}\texttt{).}\\
\texttt{fact\_alt\_tf(}\mathbf{\texttt{1}}\texttt{,A)} & \xrightarrow{\beta} \mathbf{\texttt{mult\_tf(A,1)}}\texttt{;}\\
\texttt{fact\_alt\_tf(N,A)} & \xrightarrow{\gamma} \texttt{fact\_alt\_tf(N-1,}\mathbf{\texttt{[N|A]}}\texttt{).}\\
\texttt{mult\_alt\_tf(\ \ \ [],A)} & \xrightarrow{\delta} \texttt{A;}\\
\texttt{mult\_alt\_tf([N|L],A)} & \xrightarrow{\epsilon} \texttt{mult\_alt\_tf(L,}\mathbf{\texttt{N*A}}\texttt{).}\quad \texttt{\% }\textit{Changed}
\end{array}
$$

Here, we have instead

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{fact\_alt\_tf(5)}}\\
\xrightarrow{\alpha} & \texttt{fact\_alt\_tf(4,[5])} & \\
\xrightarrow{\gamma} & \texttt{fact\_alt\_tf(3,[4,5])} & \\
\xrightarrow{\gamma} & \texttt{fact\_alt\_tf(2,[3,4,5])} & \\
\xrightarrow{\gamma} & \texttt{fact\_alt\_tf(1,[2,3,4,5])} & \\
\xrightarrow{\beta} & \texttt{mult\_alt\_tf([2,3,4,5],1)} & \\
\xrightarrow{\epsilon} & \texttt{mult\_alt\_tf([3,4,5],2*1)} & = \texttt{mult\_alt\_tf([3,4,5],2)}\\
\xrightarrow{\epsilon} & \texttt{mult\_alt\_tf([4,5],3*2)} & = \texttt{mult\_alt\_tf([4,5],6)}\\
\xrightarrow{\epsilon} & \texttt{mult\_alt\_tf([5],4*6)} & = \texttt{mult\_alt\_tf([5],24)}\\
\xrightarrow{\epsilon} & \texttt{mult\_alt\_tf([],5*24)} & = \texttt{mult\_alt\_tf([],120)}\\
\xrightarrow{\delta} & \texttt{120.} &
\end{array}
$$

We see that the multiplications are $5 \cdot (4 \cdot (3 \cdot (2 \cdot 1)))$, just as they were when rewriting `fact(5)`, but this faithfully ordered sequence comes at a cost: if the number is n, a list of length n has to be created (n rewrites) before it is traversed (n additional rewrites) to perform the multiplications. Therefore, this solution incurs a 50% slowdown, compared to `fact_tf/1` ($2n$ versus $n + 1$ exactly) and requires additional memory to hold the temporary list. Nevertheless, `fact_alt_tf/1` illustrates our point, that is, the control context can be transformed into an operation upon a linear accumulator. More precisely, there are three phases: (1) a new function is created, which calls an auxiliary function with an empty list as initial accumulator; (2) the values of the control context are pushed onto the accumulator of the call; (3) these values are popped from the accumulator in the non-recursive bodies and are applied the original control context. In the case of `fact_alt_tf/1`, the first phase is done by clause α, the second by clause γ and the third by clauses β, δ and ϵ. If the control context is an associative function or operator, then the accumulator can be specialised, like we did with `fact_tf/1`, and it may even not need to be a list if it only holds one value all along.

Joining two lists. Let us examine now `join/2` and `join_tf/2`. We had the following definitions:

```
join(   [],Q)        $\xrightarrow{\alpha}$ Q;
join([I|P],Q)        $\xrightarrow{\beta}$ [I|join(P,Q)].

join_tf(P,Q)         $\xrightarrow{\gamma}$ join(P,Q,[]).
join(   [],Q,   []) $\xrightarrow{\delta}$ Q;
join(   [],Q,[I|A]) $\xrightarrow{\epsilon}$ join([],[I|Q],A);
join([I|P],Q,    A) $\xrightarrow{\zeta}$ join(P,Q,[I|A]).
```

This transformation illustrates perfectly the general method we underline here: (1) a linear accumulator, `A`, was added in a new clause γ; (2) the values in the control context (here only `I` in clause β) were stored in it by clause ζ; (3) and later popped out to be processed as in the original control context in clause ϵ, that is, pushed on the partial result—which is `Q` because of clause δ that derives from clause α.

Filtering out the first occurrence. Let us consider in turn the transformation from `rm_fst/2` to `rm_fst_tf/2`. Here is the definition of the former, as found on page 80:

```
rm_fst(_,   []) $\xrightarrow{\alpha}$ [];
rm_fst(I,[I|L]) $\xrightarrow{\beta}$ L;
rm_fst(I,[J|L]) $\xrightarrow{\gamma}$ [J|rm_fst(I,L)].
```

and of the latter on page 85:

```
rm_fst_tf(I,L)        $\xrightarrow{\delta}$ rm_fst_tf(I,L,[]).
rm_fst_tf(_,   [],A) $\xrightarrow{\epsilon}$ rev(A);
rm_fst_tf(I,[I|L],A) $\xrightarrow{\zeta}$ rev_join(A,L);
rm_fst_tf(I,[J|L],A) $\xrightarrow{\eta}$ rm_fst_tf(I,L,[J|A]).

rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

We have here another illustration of our method, although it is a little bit obscured: (1) a list accumulator has been added in clause δ; (2) the values in the control context (only `J`) are pushed onto it in clause η; (3) these values are popped from the accumulator and the original control context they belonged to is applied to them in clauses ϵ and ζ. The usage of `rev_join/2` itself is a handy shortcut, because it was already defined and it corresponds exactly to the last step we mentioned: the values from the control context, stored in the accumulator, have to be popped (this is the first phase of the definition of `rev_join/2`) and applied to the partial result as if in the original control context, that is, pushed on it (because of clause β).

Filtering out the last occurrence. The last illustrations we have to reconsider are the definitions of `rm_lst/2` and `rm_lst_tf/2`. The former is

```
rm_lst(I,L) -> rev(rm_fst(I,rev(L))).
```

while the latter is found on page 105:

```
rm_lst_tf(I,L)      -> rev_join(L,[],I).
rev_join(   [],Q,I) -> rm_fst(I,Q,[]);
rev_join([J|P],Q,I) -> rev_join(P,[J|Q],I).
rm_fst(_,   [],A)   -> A;
rm_fst(I,[I|Q],A)   -> rev_join(Q,A);
rm_fst(I,[J|Q],A)   -> rm_fst(I,Q,[J|A]).
```

This transformation is longer than the previous ones but it also underlines our general method. Two aspects make it slightly different in appearance, though. Firstly, we applied to the first two clauses of `aux2/3` two algebraic equalities to speed up the code:

$$\texttt{rev(rev(A))} \equiv \texttt{A},$$
$$\texttt{rev(rev\_join(A,L))} \equiv \texttt{rev\_join(L,A)}.$$

Secondly, we realised that the accumulator of `rev_join/3` only contained a single item, so it was transformed further into that item. We also removed an auxiliary definition which was expanded *in situ*. Lastly, we renamed some functions and variables to enhance the readability.

Exercises. [Answers on page 372.]

1. Derive an equivalent definition `diff_tf/1` in tail from `diff/1`:

   ```
   diff([M,N]) -> M - N;
   diff([M|L]) -> M - diff(L).
   ```

   Next, make sure that the function never fails by ending instead with an atom `list_too_short`. Compare the delays of `diff/1` and `diff_tf/1`.

2. Transform the definition of `srev/1` into tail form `srev_tf/1` and give its delay.

   ```
   srev(   []) -> [];
   srev([I|L]) -> join(srev(L),[I]).
   ```

   ```
   join(   [],Q) -> Q;
   join([I|P],Q) -> [I|join(P,Q)].
   ```

3. Another idea for filtering out the last occurrence of an item in a list consists in combining a sequential search with a membership test: if an occurrence is found, it is checked whether it is the last or not by performing another sequential search *on the remaining items*. (This kind of tactic is commonly called in Computer Science a *lookahead*.) If the item is found not to be the last, it is retained in the partial result and the main sequential search is resumed, otherwise it stops and the item is left out. Implement this idea as function `rm_lst3/2` and compute its delay, in particular in its best and worst cases, as well as its average delay. Compare it to the prior equivalent functions of this section.
4. Define a function `drop/2` such that the call `drop(`L`,`n`)` is rewritten in the list made of the items of list L in the same order, except the items occurring every n positions, the first item being counted as occurring at position 1. For instance

   `drop([a,b,c,d,e,f,g,h],3)` $\twoheadrightarrow$ `[a,b,d,e,g,h];`

   `drop([a,b,c,d,e,f,g,h],1)` $\twoheadrightarrow$ `[].`

   What do you think should be done when the period is 0? What about the empty list? What if the period is greater than the length of the list? Or negative? Justify your choices and make sure no special case is left out, that is, your definition is complete. Find the delay in the worst case.
5. Check the following definitions and assess their correctness and completeness. If a definition is incorrect, provide a counter-example and explain why, in your opinion, the programmer made that mistake. If a definition is overly complex or too long, improve upon it. Check also delay, number of pushes and memory usage. Which definitions are in tail form?

   (a)
   ```
   rm_fst(I,   []) -> error;
   rm_fst(I,  [I]) -> [];
   rm_fst(A,  [I]) -> [I];
   rm_fst(I,[I|L]) -> L;
   rm_fst(A,[I|L]) -> [I|rm_fst(A,L)].
   ```

   (b)
   ```
   rm_fst(X,[X|T]) -> rm_fst(T);
   rm_fst(X,[Y|T]) -> [Y|rm_fst(X,T)];
   rm_fst(T,   []) -> [].
   rm_fst([X|T])   -> [X|rm_fst(T)];
   rm_fst(   [])   -> [].
   ```

   (c)
   ```
   rm_fst(_,   [])   -> [];
   rm_fst(I,[I|L])   -> L;
   ```

```
rm_fst(A,[I|L])   -> rm_fst(A,L,[I]).
rm_fst(I,[I|L],C) -> rev_join(C,L);
rm_fst(A,[I|L],C) -> rm_fst(A,L,[I|C]);
rm_fst(_,   [],M) -> rev(M).
```

(d)
```
rm_fst(I,L)       -> rm_fst(I,L,[]).
rm_fst(I,[I|L],A) -> join(A,L);
rm_fst(I,[J|L],A) -> rm_fst(I,L,join(A,[J])).
```

(e)
```
rm_fst(I,  [I]) -> [];
rm_fst(_,  [I]) -> [I];
rm_fst(I,[I|L]) -> rm_fst(L);
rm_fst(J,[I|L]) -> [I|rm_fst(J,L)].
rm_fst(L)       -> L.
```

6. Define a function `rm_all/2` such that `rm_all(`I`,`L`)`, where I is any expression and L an expression rewritable to a list, is rewritten into a list containing the elements of L, except all occurrences of I, in the same order. For example, `rm_all(7,[7,7,1,7,2])` $\twoheadrightarrow$ `[1,2]`. Is there a worst case, that is, a set of inputs which maximise the number of rewrites? Find the delay and the number of pushes to reach the result. Give an equivalent definition `rm_all_tf/2` in tail form and answer the same questions.

Chapter 5

Flattening

Let us undertake the task of designing a function `flat/1` such that the call `flat(`L`)`, where L is an arbitrary list, is rewritten into a list containing the same items as L, in the same order, except empty lists and all inner pushes (`|`) have been removed. If L contains no list, then `flat(`L`)` $\equiv L$. Let us review some more examples to grasp the concept.

$$\begin{aligned}
\texttt{flat([])} &\twoheadrightarrow \texttt{[]};\\
\texttt{flat([3,[]])} &\twoheadrightarrow \texttt{[3]};\\
\texttt{flat([[],[[]]])} &\twoheadrightarrow \texttt{[]};\\
\texttt{flat([5,foo,[3,[]]])} &\twoheadrightarrow \texttt{[5,foo,3]};\\
\texttt{flat([[],[1,[2,[]],4],[],x])} &\twoheadrightarrow \texttt{[1,2,4,x]}.
\end{aligned}$$

Direct approach. Let us try a direct approach. A list is either empty or not:

```
flat(   []) -> ▭;
flat([I|L]) -> ▭.
```

Then we must discriminate on the kind of head `I` because if it is not a list, for example, an integer or an atom, we keep it as it is in the result, otherwise, we must flatten it as well. Function `is_a_list/1` on page 82 shows how to distinguish a list from other kind of values. We can adapt the same idea here:

```
flat(       []) -> ▭;
flat(   [[]|L]) -> ▭;
flat([[I|M]|L]) -> ▭;
flat(    [I|L]) -> ▭.                  % I not a list
```

It is easy to guess the first and second bodies:

```
flat(       []) -> [];
flat(   [[]|L]) -> flat(L);                % Skipping []
```

`flat([[I|M]|L]) ->` $\boxed{}$;
`flat(    [I|L]) ->` $\boxed{}$.

The last clause is also easy to complete, because we know that `I` is not a list, so it can remain at the same place in the result, whilst the tail `L` is flattened:

`flat(       []) -> [];`
`flat(   [[]|L]) -> flat(L);`
`flat([[I|M]|L]) ->` $\boxed{}$;
`flat(    [I|L]) ->` **`[I|flat(L)]`**`.`   `%` *`Keeping`* `I`

The third body requires more pondering. Perhaps the trap to avoid is to think that `I` is not a list. Actually, we can know nothing of the type of `I`, although we know that `M` and `L` are lists (perhaps not flat). Different courses of action can be pursued. First, let us try to flatten `[I|M]` and `L` separately and later join the two resulting lists. Let us rename the definition to avoid confusion and tag each clause:

`sflat(       [])` $\xrightarrow{\alpha}$ `[];`
`sflat(   [[]|L])` $\xrightarrow{\beta}$ `sflat(L);`
`sflat([[I|M]|L])` $\xrightarrow{\gamma}$ **`join(sflat([I|M]),sflat(L))`**`;`
`sflat(    [I|L])` $\xrightarrow{\delta}$ `[I|sflat(L)].`

This choice is interesting because it leaves open, in theory, the possibility to compute in parallel the two recursive calls in clause γ, but the outermost call to `join/2` reminds us of the definition of `srev/1`:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

which was proved on page 69 to be quadratic in the length of the input list: the delay of `join/2` depends solely on the length of its first argument and the recursive call `srev(L)` is precisely the first argument to `join/2`, hence incurring the quadratic cost because the same items are joined a number of times proportional to the length. So, perhaps, the similarity does not run deep because, in the case of `sflat/1`, the first argument of `join/2` is the recursive call `sflat([I|M])`, which is not the tail of the list `[[I|M]|L]` but its head.

Is this enough a difference to not fear a quadratic delay? Let us try an example that exerts all the clauses of the definition. In the following, the call to be rewritten next is underlined.

$\underline{\texttt{sflat([[],[[a],b],c])}}$

$\qquad\qquad \xrightarrow{\beta} \underline{\texttt{sflat([[[a],b],c])}}$

$\qquad\qquad \xrightarrow{\gamma} \texttt{join(}\underline{\texttt{sflat([[a],b])}}\texttt{,sflat([c]))}$

$\qquad\qquad \xrightarrow{\gamma} \texttt{join(join(}\underline{\texttt{sflat([a])}}\texttt{,sflat([b])),sflat([c]))}$

$$\begin{aligned}
&\xrightarrow{\delta} \texttt{join(join([a|}\underline{\texttt{sflat([])}}\texttt{],sflat([b])),sflat([c]))}\\
&\xrightarrow{\alpha} \texttt{join(join([a|[]],sflat([b])),sflat([c]))}\\
&= \texttt{join(join([a],}\underline{\texttt{sflat([b])}}\texttt{),sflat([c]))}\\
&\xrightarrow{\delta} \texttt{join(join([a],[b|}\underline{\texttt{sflat([])}}\texttt{]),sflat([c]))}\\
&\xrightarrow{\alpha} \texttt{join(join([a],[b|[]]),sflat([c]))}\\
&= \texttt{join(join([a],[b]),}\underline{\texttt{sflat([c])}}\texttt{)}\\
&\xrightarrow{\delta} \texttt{join(join([a],[b]),[c|}\underline{\texttt{sflat([])}}\texttt{])}\\
&\xrightarrow{\alpha} \texttt{join(join([a],[b]),[c|[]])}\\
&= \texttt{join(join([a],[b]),[c])}\\
&\vdots\\
&\rightarrow \texttt{[a,b,c]}.
\end{aligned}$$

Notice that it was possible to choose to rewrite the call to `join/2` when considering `join(join([a],[b]),sflat([c]))`, but we preferred to rewrite the call to `sflat/1` because this allows us to didactically distinguish the computations of`join/2` from those of `sflat/1`. The abstract syntax tree of `join(join([a],[b]),[c])` in FIGURE 20 on page 119 brings to the fore the deep similarity with `srev/1` as shown in FIGURE 17 on page 60. The key point to notice is how a leftmost branch, that is, a series of connected edges, of `join` nodes can be produced, leading to a quadratic behaviour because the same items on the leftmost subtrees are processed several times until the root is reached.

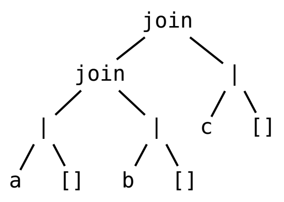


FIGURE 20: Abstract syntax tree of `join(join([a],[b]),[c])`

Lifting. There is another approach to the flattening of a list, which consists in lifting item `I` in clause γ one level up among the embedded lists, thus approaching little by little a flat list. Given a list of values written without the symbol "`|`", the *embedding level* of an item is the number of symbols "`[`" to its left, including the outermost one, minus the number of symbols "`]`" encountered. The embedding levels of items `a`, `b` and `c` in `[[],[[a,[b]]],c]` are, respectively, $4-1=3$, $5-1=4$ and $5-4=1$. This is visualised better this way (opening square brackets

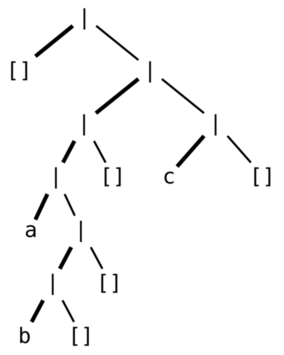


FIGURE 21: Embedding levels of 1 in bold in `[[],[[a,[b]]],c]`

in bold and item underlined):

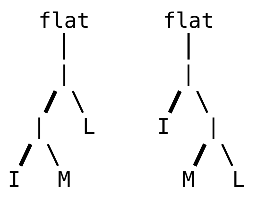
$$\underbrace{\texttt{[[],[[}}_{\mathbf{4}-1}\underline{\texttt{a}}\texttt{,[b]]],c]} \qquad \underbrace{\texttt{[[],[[a,[}}_{\mathbf{5}-1}\underline{\texttt{b}}\texttt{]]],c]} \qquad \underbrace{\texttt{[[],[[a,[b]]],}}_{\mathbf{5}-4}\underline{\texttt{c}}\texttt{]}$$

(Note that this example is different from the one we used to test `sflat/1`.) But `[a|[b|[]]]` should be written `[a,b]` first before applying the previous definition, otherwise, `b` would have a embedding level of 2 instead of the correct level 1. This is what was meant by "Given a list written *without the symbol* '`|`',[...]"

The definition allows items to be lists as well and, for instance, the embedding levels of `[]`, `[a,[b]]` and `[b]` are, respectively, $1 - 0 = 1$, $3 - 1 = 2$ and $4 - 1 = 3$:

$$\underbrace{\texttt{[}}_{\mathbf{1}-0}\underline{\texttt{[]}}\texttt{,[[a,[b]]],c]} \qquad \underbrace{\texttt{[[],[}}_{\mathbf{3}-1}\underline{\texttt{[a,[b]]}}\texttt{],c]} \qquad \underbrace{\texttt{[[],[[a,}}_{\mathbf{4}-1}\underline{\texttt{[b]}}\texttt{]],c]}$$

By *lifting* we meant that the embedding level of item `I` was reduced, more precisely it is decremented by one. The embedding level of an item can be alternatively seen on the abstract syntax tree as the number of left-oriented edges in the path from the root to the node containing the item in question. See, for example, FIGURE 21 on page 120 where left-oriented edges are in bold and each of them corresponds to one level of embedding on a path up to the root. How to lift exactly one item up one level? The general scheme is shown in FIGURE 22 on page 121, where it is evident that the embedding level of `I` is 3 and becomes 2 after the rewrite, whilst the levels of `M` and `L` remain unchanged—respectively, 1 and 0. It is worth noting that the representation of Erlang expressions as abstract syntax trees enables a natural extension of the concept of embedding level to sub-lists, like `L`: a sub-list has the same embedding level as the embedding list because the edge from a list to its immediate



(a) Before (b) After

FIGURE 22: Lifting `I` only one level up

sub-list is, by construction, right-oriented. When a list is an item of another list, that is, it is graphically connected downwardly by a left-oriented edge, it is said to be *embedded*, and it is not a sub-list. For instance, in the value `[[]]`, the list `[]` is embedded, while the implicit `[]` terminating the whole list, visible if we write instead `[[]|**[]**]`, is a sub-list. The definition corresponding to the lifting technique, translated from FIGURE 22 on page 121, is

`flat(       [])` $\xrightarrow{\alpha}$ `[];`
`flat(   [[]|L])` $\xrightarrow{\beta}$ `flat(L);`
`flat([[I|M]|L])` $\xrightarrow{\gamma}$ **`flat([I,M|L]);`** `% I` *`lifted one level up`*
`flat(    [I|L])` $\xrightarrow{\delta}$ `[I|flat(L)].`

By repeating clause η, the first item of the first item etc. of `I` (following the leftmost path from the root to a leaf) will end up first of the outermost list and will be, in turn, matched against the empty list (clause ζ) or a non-list (clause θ). Let us run the same example again:

`flat([[],[[a,[b]]],c])` $\xrightarrow{\beta}$ `flat([[[a,[b]]],c])`
$\xrightarrow{\gamma}$ `flat([[a,[b]],[],c])`
$\xrightarrow{\gamma}$ `flat([a,[[b]],[],c])`
$\xrightarrow{\delta}$ `[a|flat([[[b]],[],c])]`
$\xrightarrow{\gamma}$ `[a|flat([[b],[],[],c])]`
$\xrightarrow{\gamma}$ `[a|flat([b,[],[],[],c])]`
$\xrightarrow{\delta}$ `[a|[b|flat([[],[],[],c])]]`
$=$ `[a,b|flat([[],[],[],c])]`
$\xrightarrow{\beta}$ `[a,b|flat([[],[],c])]`
$\xrightarrow{\beta}$ `[a,b|flat([[],c])]`
$\xrightarrow{\beta}$ `[a,b|flat([c])]`
$\xrightarrow{\delta}$ `[a,b|[c|flat([])]]`
$=$ `[a,b,c|flat([])]`
$\xrightarrow{\alpha}$ `[a,b,c|[]]`
$=$ `[a,b,c].`

These rewrites should also be seen in two dimensions, that is, on the abstract syntax trees shown in FIGURES 23 to 24 on pages 123–124, where the original empty list is inscribed in a dotted circled to distinguish it from the empty lists used to build non-empty lists, and the root of the subtree being rewritten next is boxed as usual (here, only `flat`).

Delays. Let us now compare `sflat/1` and `flat/1` in terms of the number of rewrites to reach a value—in other words, their delays. We usually take as measure of a list its length, but this approach does not work here. For instance, `[[1,w],6]` and `[a,b]` have the same length, but the delay of their flattening differ. Examining how many times each clause is used gives us a clue about the right measures needed to assess the delay for the whole definition. Starting with `sflat(`L`)`, where L is an arbitrary list, and recalling the definition on page 118

```
sflat(       []) -α-> [];
sflat(   [[]|L]) -β-> sflat(L);
sflat([[I|M]|L]) -γ-> join(sflat([I|M]),sflat(L));
sflat(    [I|L]) -δ-> [I|sflat(L)].
```

it comes that

- clause β is used once for each empty list originally in the input;
- clause α is used once when the end of the input is reached *and* once for each empty list generated by the call `sflat(L)` in clause γ;
- clause δ is used once for each item which is not a list;
- clause γ is used once for each non-empty embedded list.

We need to add the delay of calling `join/2` in clause γ but, first, let us realise that we now know the parameters which the delay depends upon:

1. the number of items which are not lists, that is, `len(sflat(`L`))`;
2. the number of non-empty lists embedded in the input, say $\mathcal{N}$;
3. the number of original empty lists, say $\mathcal{E}$.

Then we can reformulate the above analysis in the following terms:

- clause β is used $\mathcal{E}$ times;
- clause α is used $1+\mathcal{N}$ times;
- clause δ is used `len(sflat(`L`))` times;
- clause γ is used $\mathcal{N}$ times.

So the delay due to the clauses of `sflat/1` alone is $1+$`len(sflat(`L`))`$+\mathcal{E}+2\cdot\mathcal{N}$. Hence, for instance, in the case of `sflat([[],[[a],b],c])`, we should expect $1+3+1+2\cdot 2 = 9$ rewrites—which is correct, according to our previous rewrites on page 118. Now we must add the delay of calling `join/2`. We already know on page 66 that it is the length of its

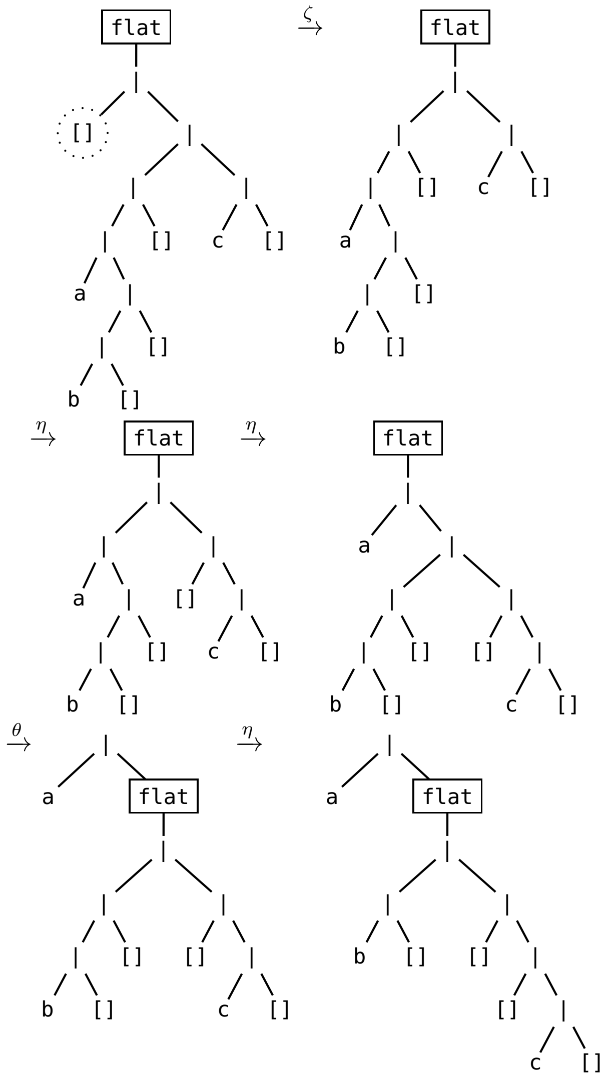


FIGURE 23: `flat([[],[[a,[b]]],c])` $\twoheadrightarrow$ `[a|flat([[b],[],[],c])]`

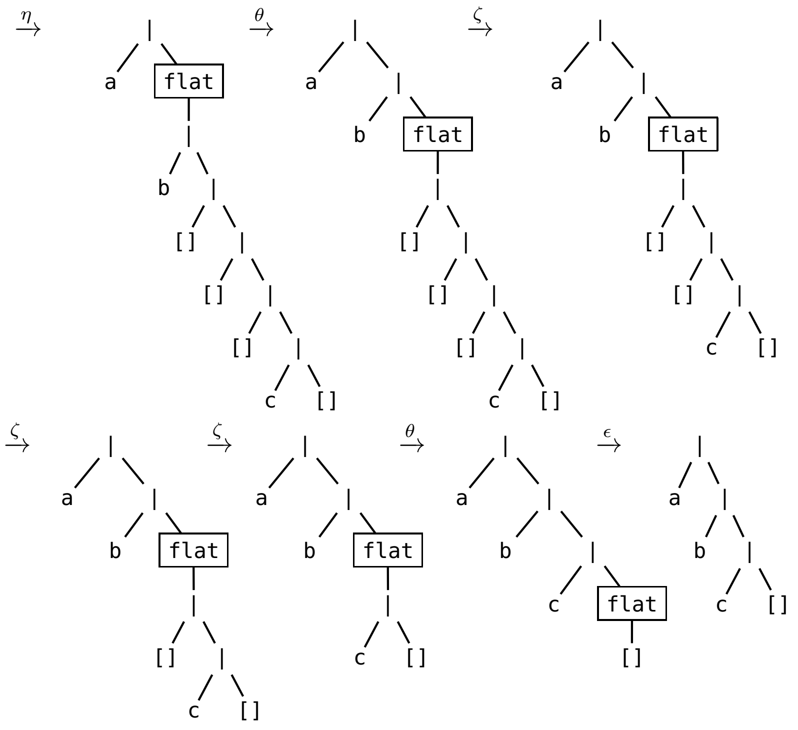


FIGURE 24: `[a|flat([b,[],[],[],c])]` $\twoheadrightarrow$ `[a,b,c]`

first argument plus one, formally: $\mathcal{D}_n^{\texttt{join}} = n + 1$. Clause γ shows that `join/2` is called on all flattened non-empty embedded lists, so the total delay of these calls is the number of non-list items in all embedded lists, plus the number of embedded lists (this is due to the "+1" in $n + 1$). The latter number is none other than $\mathcal{N}$ but the former is perhaps more obscure.

For a better understanding, let us visualise in FIGURE 25 on page 125 the abstract syntax tree of some list like `[[],[[a],b],c]`. The circled nodes corresponds to *list constructors* (also known as *pushes*, symbolised by `(|)`) of embedded lists. Next to them is the number of non-list items in the tree rooted at the node. Hence 1 because of the unique item `a` (clause ϵ) and 2 because of `b` and `a` *again* (ζ). It is convenient to visualise these numbers going up from the leaves and being added at circled nodes. Let us note $\mathcal{P}$ the number we try to understand: it is the sum of all the numbers at the circled nodes, in our example, it is

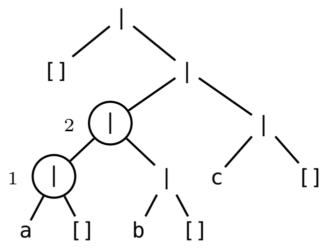


FIGURE 25: Abstract syntax tree of `[[],[[a],b],c]`

thus $1 + 2 = 3$. So the total delay of `sflat(`L`)` is

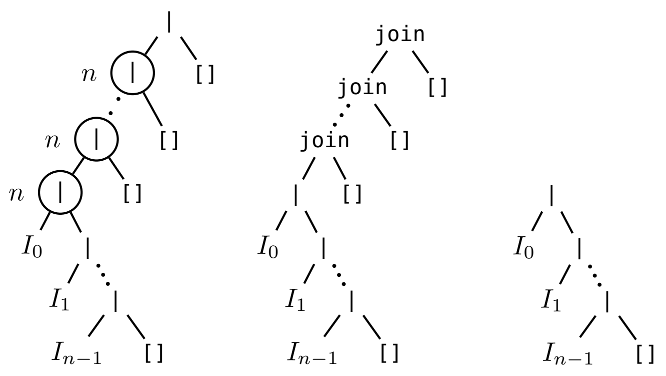
$$1 + n + \mathcal{E} + 3 \cdot \mathcal{N} + \mathcal{P},$$

where the length of `sflat(`L`)` is n. In our example, $\mathcal{P} = 3$, $\mathcal{E} = 1$ (one empty list in the input), $\mathcal{N} = 2$ (two non-empty embedded lists, at the circled nodes) and the length of `sflat([[],[[a],b],c])` is 3. So, the total delay, including the rewrites by `join/2` is $1 + 3 + 1 + 3 \cdot 2 + 3 = 14$. Indeed, we found that the delay for `sflat/1` calls alone was 9, and 14 is consistent with the fact that 5 additional steps are required to evaluate `join(join([a],[b]),[c])`. Unfortunately, it is almost as difficult to express algorithmically $\mathcal{P}$ as it is to define `sflat/1`, so it is certainly worth finding another, more intuitive, point of view.

The idea may rise from the observation made in passing above, when computing $\mathcal{P}$ on the example of FIGURE 25 on page 125: a circled node was annotated with ζ because it counts `a` *twice*. In other words, the atom item `a` is counted for one at each circled node because it is included in two non-empty embedded lists, one of which is included in the other: `a` is embedded in `[a]`, which is, in turn, embedded in `[[a],b]`. Therefore, if we fix the size of the output, that is, the number of non-list items, we can wonder how to arrange these items so as to maximise the delay. This is an example of worst case analysis which is based on the output size instead of the input size. When the delay depends on the result of the call, it is said to be *output-dependent*. If we want to count as many times as possible the same items, we should put all of them in the deeper leftmost subtree. That is, we set the input

$$L = \texttt{[}\ \underbrace{\texttt{[...[}}_{\mathcal{N}\text{ times}}\ I_0, I_1, \ldots, I_{n-1}\ \underbrace{\texttt{]...]}}_{\mathcal{N}\text{ times}}\ \texttt{]},$$

where the I_i are non-list items. Here, the number of items which are non-lists, called $\mathcal{N}$, is none other than the embedding level of the non-list items I, minus one. Then $\mathcal{P} = n \cdot \mathcal{N}$. Consider FIGURE 26 on page 126, where (1) FIGURE 26a on page 126 is the input in the worst



(a) Input L to `sflat/1` (b) Partial result (c) Result

FIGURE 26: `sflat([...[`$I_0, I_1, \ldots, I_{n-1}$`]...])` $\twoheadrightarrow$ `[`$I_0, I_1, \ldots, I_{n-1}$`]`

case, whose embedded lists are annotated with the number of non-list items they contain, as in FIGURE 25 on the facing page; (2) FIGURE 26b on the next page displays the abstract syntax tree after all calls to `sflat/1` on the input have been rewritten, hinting at the total number of steps required to finish the computation: $\mathcal{N} \cdot (n+1)$. (Let us not forget that this is not the way Erlang computes, so this presentation is set out as a didactic mean only.) and (3) FIGURE 26c on the following page shows the final result. Of course, it is possible to increase the number of rewrites by inserting empty lists everywhere in the input, as this triggers clause β but this case is already accounted for. As a conclusion, the delay in the worst case of `sflat(`L`)` is

$$\mathcal{W}^{\mathsf{sflat}}_{n,\mathcal{N}} = 1 + n + 2 \cdot \mathcal{N} + \mathcal{N} \cdot (1+n) = (\mathcal{N}+1)(n+3) - 2,$$

where n is the length of `sflat(`L`)` and $\mathcal{N}$ is the number of embedded non-lists. All the terms involved in expressing the delay are now intuitive. For example `sflat([[[[[[a,b]]]]]])` has a delay of $(4+1)(2+3)-2 = 23$ rewrites.

Let us consider now the delay of `flat(`L`)`, where L is an arbitrary list. We settled for the following definition:

```
flat(        []) -α-> [];
flat(    [[]|L]) -β-> flat(L);
flat([[I|M]|L]) -γ-> flat([I,M|L]);
flat(     [I|L]) -δ-> [I|flat(L)].
```

We draw that

- clause α is used once when the end of the input is reached;
- clause γ is used once for each item of all the embedded lists;

- clause β is used once for each empty list originally in the input *and* once for each empty list `M` by clause γ (if `I` is empty, it has already been accounted for as an empty list originally present);
- clause δ is used once for each item which is not a list.

Therefore, the delay is

$$1 + \mathcal{L} + (\mathcal{E} + \mathcal{N}) + n.$$

where $\mathcal{L}$ is the sum of the lengths of all the sub-lists (clause γ), $\mathcal{E}$ is the number of empty lists originally in the input (clause β), $\mathcal{N}$ is the number of non-empty embedded lists (clause β) and n is the number of non-list items (clause δ). For example,

$$\llbracket\texttt{flat([[[[[a,b]]]]])}\rrbracket = 1 + 5 + (0 + 4) + 2 = 12,$$
$$\llbracket\texttt{flat([[],[[a],b],c])}\rrbracket = 1 + 3 + (1 + 2) + 3 = 10,$$
$$\llbracket\texttt{flat([[],[[a,[b]]],c])}\rrbracket = 1 + 4 + (1 + 3) + 3 = 12.$$

These are the expected values and the last case was laid out in details on page 121.

How does `flat/1` compare with `sflat/1` in general? The delay of calling `sflat/1` is $1 + n + \mathcal{E} + 3 \cdot \mathcal{N} + \mathcal{P}$, so the question boils down to comparing $2 \cdot \mathcal{N} + \mathcal{P}$ with $\mathcal{L}$. Let us consider a few examples:

- Let the input be `[a,b]`. Then $\mathcal{N} = 0$, $\mathcal{P} = 0$ and $\mathcal{L} = 0$, so $2{\cdot}0{+}0 = 0$; that is, the delays are equal.
- Let the input be `[[a,b]]`. Then $\mathcal{N} = 1$, $\mathcal{P} = 2$ and $\mathcal{L} = 2$, so $2 \cdot 1 + 2 > 2$; that is, `flat/1` is faster than `sflat/1`.
- Let the input be `[[[],[],[]]]`. Then $\mathcal{N} = 1$, $\mathcal{P} = 0$ and $\mathcal{L} = 3$, so $2 \cdot 1 + 0 < 3$; that is, `sflat/1` is faster than `flat/1`.

Therefore, both functions are not comparable in general. Nevertheless, if we have no empty list in the input, that is, if $\mathcal{E} = 0$, then $\mathcal{P} \geqslant \mathcal{L}$, because the length of each embedded list is lower than or equal to the number of non-list items at greater or equal embedding levels (it is equal if there is no further embedded list). Therefore, summing up all these inequalities, we draw that if $\mathcal{E} = 0$, then `flat/1` is faster than `sflat/1`.

Slight improvement. Is it possible to improve the definition of `flat/1`? Upon close examination of the example given in FIGURES 23 to 24 on pages 123–124, it becomes apparent that, for each non-empty embedded list an empty list is produced when lifting the last item in them (`M` is an empty list then). These additional empty lists have to be removed by means of clause β. It is easy to avoid introducing empty lists which need to be eliminated next: let us just add a clause matching the case of an embedded list with one item, that is, a singleton list:

```
flat_opt(       []) -> [];
flat_opt(   [[]|L]) -> flat_opt(L);
flat_opt(  [[I]|L]) -> flat_opt([I|L]);            % Improvement
flat_opt([[I|M]|L]) -> flat_opt([I,M|L]);                 % M ≠ []
flat_opt(    [I|L]) -> [I|flat_opt(L)].
```

What is the delay of this improved definition? Simply, the term $\mathcal{N}$ vanishes:

$$1 + \mathcal{L} + \mathcal{E} + n.$$

As a consequence, we now have the delays

$$[\![\texttt{flat\_opt([[[[[a,b]]]]])}]\!] = 1 + 5 + 0 + 2 = 8,$$
$$[\![\texttt{flat\_opt([[],[[a],b],c])}]\!] = 1 + 3 + 1 + 3 = 8,$$
$$[\![\texttt{flat\_opt([[],[[a,[b]]],c])}]\!] = 1 + 4 + 1 + 3 = 9.$$

The improvement goes beyond a shorter delay: the delay itself can be expressed with one notion less—namely here, the number of embedded lists, noted $\mathcal{N}$. This is important too.

Exercises.

1. Reconsider the delay of `sflat/1` if we augment the definition of `join/2` with a special case (in bold):

   ```
   join(    P,[]) -> P;
   join(   [], Q) -> Q;
   join([E|P], Q) -> [E|join(P,Q)].
   ```

2. Check that the following definition is equivalent to `flat/1` and find its delay.

   ```
   flat_bis(L)          -> flat_bis(L,[],[]).
   flat_bis(   [],[],B) -> rev(B);
   flat_bis(   [], A,B) -> flat_bis(A,   [],    B);
   flat_bis(  [I], A,B) -> flat_bis(I,    A,    B);
   flat_bis([I|L], A,B) -> flat_bis(I,[L|A],    B);
   flat_bis(    I, A,B) -> flat_bis(A,   [],[I|B]).
   ```

Chapter 6

Delay and Tail Form Revisited

When determining the delay of a function, we often end up with recurrence equations to solve. There are many methods to deal with special and common systems exactly, but it is sometimes possible to rely on some heuristics as well. For instance, let us recall equations (3.4) on page 69:

$$\mathcal{D}_0^{\mathsf{srev}} = 1; \quad \mathcal{D}_n^{\mathsf{srev}} = 1 + n + \mathcal{D}_{n-1}^{\mathsf{srev}}, \text{ where } n > 0. \tag{6.11}$$

If we have the intuition that the delay is quadratic, then it can be determined within a few tries. Let

$$\mathcal{D}_n^{\mathsf{srev}} := an^2 + bn + c, \text{ with } a, b, c \in \mathbb{Q},$$

and let us find the values of the coefficients. Since there are three of them, we need at least three values of $\mathcal{D}_n^{\mathsf{srev}}$ to solve the linear equations

$$\begin{aligned}
\mathcal{D}_0^{\mathsf{srev}} &= 1 = c,\\
\mathcal{D}_1^{\mathsf{srev}} &= 1 + 1 + 1 = 3 = a + b + c,\\
\mathcal{D}_2^{\mathsf{srev}} &= 1 + 2 + 3 = 6 = a \cdot 2^2 + b \cdot 2 + c.
\end{aligned}$$

Therefore, replacing c by its value, we draw $c = 1$, $a + b = 2$ and $4a + 2b = 5$; finally $a = 1/2$, $b = 3/2$ and $c = 1$, that is $\mathcal{D}_n^{\mathsf{srev}} = (n^2 + 3n + 2)/2 = (n + 1)(n + 2)/2$, as we found on page 69. Since the assumption about the quadratic behaviour could have been wrong, it is then important to try other values with the newly found formula, for instance $\mathcal{D}_4^{\mathsf{srev}} = (4 + 1)(4 + 2)/2 = 15$, then compare with the count of all the rewrites to compute the call `srev([a,1,7,a])`, for example. Here, the contents of the list is irrelevant, only its length is. After finding a formula for the delay using the empirical method above, it is necessary to prove it for all values of n. Since the initial equations are recurrent, the proof method of choice is the *proof by induction*. In general, let us suppose we want to prove a property $\mathcal{P}$ of n, written $\mathcal{P}(n)$. The first step consists in verifying that the property holds for the smallest value of n, let us say $\mathcal{P}(0)$. The second step postulates $\mathcal{P}(n)$ for some given n (this is called the *induction hypothesis*). Finally, we prove $\mathcal{P}(n+1)$. The *induction principle* then allows us to conclude that $\mathcal{P}(n)$ is true for *any* n. In our example, we want to prove the property $\mathcal{P}(n)$ stating that $\mathcal{D}_n^{\mathsf{srev}} = (n^2+3n+2)/2$. By definition of the empirical approach that lead to the formula, we already checked its validity for some small values, here, $n = 0, 1, 2$. Let us suppose it valid for some value of n and let us prove $\mathcal{D}_{n+1}^{\mathsf{srev}} = ((n+1)^2+3(n+1)+2)/2$. Equation (6.11) implies

$$\mathcal{D}_{n+1}^{\mathsf{srev}} = 1 + (n+1) + \mathcal{D}_n^{\mathsf{srev}}.$$

The induction hypothesis, in turn, implies

$$\begin{aligned}\mathcal{D}_{n+1}^{\mathsf{srev}} &= 2 + n + (n^2+3n+2)/2 = (n^2+5n+6)/2\\ &= ((n+1)^2+3(n+1)+2)/2,\end{aligned}$$

which is $\mathcal{P}(n+1)$. Therefore, the induction principle says that the delay we found experimentally is always correct.

The same technique can be applied to compute $\mathcal{D}_n^{\mathtt{rm\_fst\_alt}}$, on page 94, for example, or any other function for which we have some recurrence equations defining its delay and about which we suppose the solved form is of a specific function in terms of n.

It is inconvenient and error-prone to rewrite by hand the calls we need to set the linear equations. Why use paper and pencil when we can use Erlang instead? The idea is to transform a given function definition into another which computes the same values *and* the number of rewrites to reach it. It seems then a good idea to change the final values to be a list containing these two pieces of data, but a list supposes that it is always possible to add (push) or subtract (pop) some information from it, whereas what we want here is a fixed, closes, number of values packaged together. In order to do this, we need a new concept: the *tuple*. Erlang tuples are conceptually like mathematical tuples, except that they are written differently. For instance, in mathematics, we would write $(4,7)$, whilst in Erlang this same tuple is written `{4,7}`. In other words, Erlang uses curly braces in stead of parentheses. Special cases of tuples are *pairs*, which compound exactly two values, like `{4,7}`, and *triples*, which gather exactly three values. Another way to conceive tuples is to think them as a kind of list on which no item can be pushed: it must be defined by giving all its items at once.

Now how do we proceed? Let us take again the definition of `rm_lst_tf/2`:

```
rm_lst_tf(I,L)      -> rev_join(L,[],I).
rev_join(   [],Q,I) -> rm_fst(I,Q,[]);
rev_join([J|P],Q,I) -> rev_join(P,[J|Q],I).
rm_fst(_,   [],A)   -> A;
rm_fst(I,[I|Q],A)   -> rev_join(Q,A);
rm_fst(I,[J|Q],A)   -> rm_fst(I,Q,[J|A]).
```

The methods consists in adding a parameter which is a counter, adding an initial call which sets it to `0` or `1` and incrementing its value in each clause (changes in bold):

```
rm_lst_tf(I,L)        -> rev_join(I,L,[],1).              % One
rev_join(   [],Q,I,C) -> rm_fst(I,Q,[],C+1);
rev_join([J|P],Q,I,C) -> rev_join(P,[J|Q],I,C+1).
rm_fst(_,   [],A,C)   -> {A,C+1};              % Result is a pair
rm_fst(I,[I|Q],A,C)   -> rev_join(Q,A,C+1);
rm_fst(I,[J|Q],A,C)   -> rm_fst(I,Q,[J|A],C+1).
```

This simple and effective approach can be implemented with all definitions in tail form.

Let us consider what happens with a definition which is not in tail form, for instance, `flat_opt/1` on the current page:

```
flat_opt(       []) -> [];
flat_opt(   [[]|L]) -> flat_opt(L);
flat_opt(  [[I]|L]) -> flat_opt([I|L]);
flat_opt([[I|M]|L]) -> flat_opt([I,M|L]);
flat_opt(    [I|L]) -> [I|flat_opt(L)].
```

If we try the previous direct manner, we need to (1) add a counter as an extra parameter everywhere in the clauses of `flat_opt/1`, which becomes `flat_opt/2`; (2) add a definition for `flat_opt/1` setting the counter's initial value; (3) make sure to increment the counter in all the calls and (4) for each body which is not a call, pair up the expression with the counter plus one (changes in bold):

```
flat_opt(L)           -> flat_opt(L,0).                   % Not 1
flat_opt(       [],C) -> {[],C+1};
flat_opt(   [[]|L],C) -> flat_opt(L,C+1);
flat_opt(  [[I]|L],C) -> flat_opt([I|L],C+1);
flat_opt([[I|M]|L],C) -> flat_opt([I,M|L],C+1);
flat_opt(    [I|L],C) -> [I|flat_opt(L,C+1)].           % Problem
```

The problem lies with the last clause, which is not in tail form: the value of the call `flat_opt(L,C+1)` is a pair, but it is treated as a list because the control context is `[I|␣]`. We need a way to hold the value of the call, then extract the list from it, push item `I` on it and put back the augmented list together with the counter in a new pair. All this can be done by simply adding an auxiliary function, say aux/2:

```
flat_opt(L)           -> flat_opt(L,0).
flat_opt(       [],C) -> {[],C+1};
flat_opt(   [[]|L],C) -> flat_opt(L,C+1);
flat_opt(  [[I]|L],C) -> flat_opt([I|L],C+1);
flat_opt([[I|M]|L],C) -> flat_opt([I,M|L],C+1);
flat_opt(    [I|L],C) -> aux(I,flat_opt(L,C+1)).
aux(I,{L,C})          -> {[I|L],C}.                 % C unchanged
```

Notice how the counter C is left invariant in the definition of aux/2 because we only want to measure the delay of flat_opt/2 (that is precisely why we initialised the counter to 0 instead of 1 in the clause defining flat_opt/1).

What about the following definition?

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

The definition is not in tail form, hence we expect to add an auxiliary function, aux/2, to handle the call to join/2, and an auxiliary function srev/2 to introduce the accumulator:

```
srev(L)       -> srev(L,0).
srev(   [],C) -> {[],C};
srev([I|L],C) -> aux(srev(L,C+1),I).
aux({L,C},I)  -> {join(L,[I]),C}.
```

If we want to also count the rewrites by join/2, a counter must also be added to the definition of join/2. Let us take the definition

```
join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

and follow the same method of transformation:

```
join(   [],Q,C) -> {Q,C+1};
join([I|P],Q,C) -> aux1(I,join(P,Q,C+1)).
aux1(I,{L,C})   -> {[I|L],C}.
```

Now the call to join/3 in the body of aux/2 is in tail form:

```
aux({L,C},I)    -> join(L,[I],C).
```

Note how join/3 takes care of constructing a pair, so there is no point in making another one in the body of aux/2.

Let us consider now a definition more involved, like the following on page 118:

```
sflat(       []) -> [];
sflat(   [[]|L]) -> sflat(L);
sflat([[I|M]|L]) -> join(sflat([I|M]),sflat(L));
sflat(    [I|L]) -> [I|sflat(L)].
```

The difficulty here lies in the third clause, because it contains two recursive calls. Since the order of argument evaluation is not specified in Erlang, we do not know which call is computed first. Therefore there are two approaches to add a rewrite counter: either keeping the indeterminacy or forcing an order. Let us first proceed with the former plan. We add a counter to the definition and add another definition whose purpose is to initialise the counter:

```
sflat1(L)          -> sflat(L,0).
sflat(       [],C) -> {[],C+1};
sflat(   [[]|L],C) -> sflat(L,C+1);
sflat([[I|M]|L],C) ->
                       join(sflat([I|M],□),sflat(L,□),□);
sflat(    [I|L],C) -> [I|sflat(L,C+1)].
```

The last clause can be handled as we did before with `flat_opt/2`:

```
sflat1(L)          -> sflat(L,0).
sflat(       [],C) -> {[],C+1};
sflat(   [[]|L],C) -> sflat(L,C+1);
sflat([[I|M]|L],C) ->
                       join(sflat([I|M],□),sflat(L,□),□);
sflat(    [I|L],C) -> aux(I,sflat(L,C+1)).
aux(I,{L,C})       -> {[I|L],C}.
```

In the third clause, we call recursively with a new counter set to 0, so we need an auxiliary to sum these two resulting counters to `C` and call `join/3`:

```
sflat1(L)          -> sflat(L,0).
sflat(       [],C) -> {[],C+1};
sflat(   [[]|L],C) -> sflat(L,C+1);
sflat([[I|M]|L],C) -> aux1(sflat([I|M],0),sflat(L,0),C+1);
sflat(    [I|L],C) -> aux(I,sflat(L,C+1)).
aux(I,{L,C})            -> {[I|L],C}.
aux1({L1,C1},{L2,C2},C) -> join(L1,L2,C1+C2+C).
```

The second approach consists in imposing an order of evaluation on the two recursive calls. Let us say, arbitrarily, that we compute `sflat([I|M])` before `sflat(L)`. Let us resume the transformation at the following stage (we rename `flat/1` into `flat2/1` to avoid confusion with the first approach):

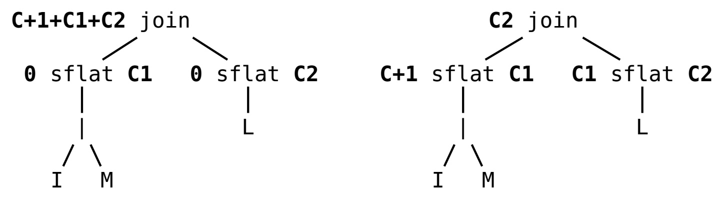
```
sflat2(L)          -> sflat(L,0).
```

```
sflat(        [],C) -> {[],C+1};
sflat(    [[]|L],C) -> sflat(L,C+1);
sflat([[I|M]|L],C) ->
                       aux1(sflat([I|M],▭),sflat(L,▭),▭);
sflat(     [I|L],C) -> aux(I,sflat(L,C+1)).
aux(I,{L,C})           -> {[I|L],C}.
aux1({L1,C1},{L2,C2},C) -> join(L1,L2,C1+C2+C).
```

The idea then consists, in the third clause of `sflat/2`, in passing the counter `C+1` to the call `sflat([I|M],▭)`, then extract from the result the new counter and pass it to the second call, `sflat(L,▭)`. Function `aux1/1` must be modified and an auxiliary function `aux2/1` is needed:

```
sflat2(L)           -> sflat(L,0).
sflat(        [],C) -> {[],C+1};
sflat(    [[]|L],C) -> sflat(L,C+1);
sflat([[I|M]|L],C) -> aux1(L,sflat([I|M],C+1));
sflat(     [I|L],C) -> aux(I,sflat(L,C+1)).
aux(I,{L,C})        -> {[I|L],C}.
aux1(L,{P,C})       -> aux2(P,sflat(L,C)).
aux2(P,{Q,C})       -> join(P,Q,C).
```

Notice how the auxiliary functions keep the counter invariant because they are not part of the original design of `sflat/1`. On the one hand, the definition of `sflat2/2` is longer than that of `sflat1/2` and it does not keep the order of evaluation undetermined. On the other hand, `sflat2/2` is more efficient because it does not set two counters to zero when matching an embedded non-empty list, these zeros being later uselessly added. If we note $\mathcal{N}$ the number of non-empty embedded lists in the original input, the third clause of `sflat1/2` is used $\mathcal{N}$ times, which means that $2\mathcal{N}$ useless additions to zero are performed. For inputs with a small number of non-empty embedded lists, this probably will make no notable difference because the delay of `sflat/1` is linear in $\mathcal{N}$, but this is enough ground to deem `sflat2/2` faster than `sflat1/1`. The different strategies can be summed up in FIGURE 27, where the abstract syntax tree of `join(sflat([I|M]),sflat(L))` is presented. Each node is annotated on its left by the value of the counter just before the corresponding call is computed (counter in). The annotations on the right-side of the nodes corresponds to the value of the counter after the corresponding call is just finished being computed (counter out). Of course, it is `sflat1/2` and `sflat2/2` which are called with a counter, instead of `sflat/1`, that is why the counters are annotations to the abstract syntax tree and are not depicted as arguments. FIGURE 27a on page 135 shows that each recursive call to `sflat/1` is done with



(a) Parallel (`sflat1/1`) (b) Sequential (`sflat2/1`)

FIGURE 27: Two ways of computing the delay of `sflat([[I|M]|L])`

a counter set to 0 and each resulting counters, `C1` and `C2`, are then added to `C+1` to give the counter finally passed to `join/3`. FIGURE 27b on page 135 demonstrates that the current counter, `C+1`, is passed to `sflat([I|M])` first, thus imposing an order of evaluation on the recursive calls; the resulting counter `C1` is, in turn, passed to `sflat(L)` and the subsequent counter, `C2`, becomes the counter finally passed to `join/3`. Let us consider one more case, function `rm_fst_alt/2`, on page 92 (we rename the arrows here):

```
rm_fst_alt(I,L)        $\xrightarrow{\beta}$ rm_fst__(I,L,[]).
rm_fst__(_,   [],A)    $\xrightarrow{\gamma}$ A;
rm_fst__(I,[I|L],A)    $\xrightarrow{\delta}$ join(A,L);
rm_fst__(I,[J|L],A)    $\xrightarrow{\epsilon}$ rm_fst__(I,L,join(A,[J])).
```

We know from the previous example that we can reuse `join/3` to manage the counter while joining two lists at the same time. In clause ϵ, we need to recover the counter computed by `join/3` so an extended `rm_-fst__/4` take it in turn, which means that some auxiliary function `aux/3` is needed. Before adding a counter, let us add the usage of `aux/3`:

```
rm_fst_alt(I,L)        $\xrightarrow{\beta}$ rm_fst__(I,L,[]).
rm_fst__(_,   [],A)    $\xrightarrow{\gamma}$ A;
rm_fst__(I,[I|L],A)    $\xrightarrow{\delta}$ join(A,L);
rm_fst__(I,[J|L],A)    $\xrightarrow{\epsilon}$ aux(I,L,join(A,[J])).
aux(I,L,P)             $\xrightarrow{\zeta}$ rm_fst__(I,L,P).
```

This program is clearly equivalent to the original one, as `aux/3` is the identity function for now. Let us then replace calls to `join/2` by calls to `join/3`, as announced, add the counters and then the value computed by all functions have to be a pair made of the value as calculated by the original function and the counter.

```
rm_fst_alt(I,L)          $\xrightarrow{\alpha}$ rm_fst_alt(I,L,□).
rm_fst_alt(I,L,C)        $\xrightarrow{\beta}$ rm_fst__(I,L,[],C).
rm_fst__(_,   [],A,C)    $\xrightarrow{\gamma}$ A                        % C unused yet
rm_fst__(I,[I|L],A,C) $\xrightarrow{\delta}$ join(A,L,C);
rm_fst__(I,[J|L],A,C) $\xrightarrow{\epsilon}$ aux(I,L,join(A,[J],C)).
aux(I,L,P)            $\xrightarrow{\zeta}$ rm_fst__(I,L,P).          % Type error
```

Using `join/3` instead of `join/2` lead to a change in the nature of the third argument of `aux/3`, as calls to `join/3` rewrite to a pair made of a list and an integer. So `P` in clause ζ is a pair now and the call to `rm_-fst__/3` is no longer correct. The parameter `P` needs to be destructured in the pattern so its two components, list and counter, are distinguished and the proper call to `rm_fst__/4` is set:

```
rm_fst_alt(I,L)       $\xrightarrow{\alpha}$ rm_fst_alt(I,L,□).
rm_fst_alt(I,L,C)     $\xrightarrow{\beta}$ rm_fst__(I,L,[],C).
rm_fst__(_,   [],A,C) $\xrightarrow{\gamma}$ A                       % C unused yet
rm_fst__(I,[I|L],A,C) $\xrightarrow{\delta}$ join(A,L,C);
rm_fst__(I,[J|L],A,C) $\xrightarrow{\epsilon}$ aux(I,L,join(A,[J],C)).
aux(I,L,{P,C})        $\xrightarrow{\zeta}$ rm_fst__(I,L,P,C).
```

Now, let us increment `C` where necessary, so as to account for one more function call *in the original program*:

```
rm_fst_alt(I,L)       $\xrightarrow{\alpha}$ rm_fst_alt(I,L,0).
rm_fst_alt(I,L,C)     $\xrightarrow{\beta}$ rm_fst__(I,L,[],C+1).
rm_fst__(_,   [],A,C) $\xrightarrow{\gamma}$ A                       % C unused yet
rm_fst__(I,[I|L],A,C) $\xrightarrow{\delta}$ join(A,L,C+1);
rm_fst__(I,[J|L],A,C) $\xrightarrow{\epsilon}$ aux(I,L,join(A,[J],C+1)).
aux(I,L,{P,C})        $\xrightarrow{\zeta}$ rm_fst__(I,L,P,C).
```

Notice how the counter `C` is left unchanged in clause ζ because there is no equivalent clause in the original definition, so this function call must not be accounted for. Similarly, in clause α, the initial value of the counter is `0`, because this is a newly added clause. Clause γ is still in need of attention. It is the last step of `rm_fst__/4`, so we need to pair the original result, that is, `A`, with the newly added counter `C`, properly incremented, of course. Doing so will change the type of the calls to `rm_fst_alt/4`:

```
rm_fst_alt(I,L)       $\xrightarrow{\alpha}$ rm_fst_alt(I,L,0).
rm_fst_alt(I,L,C)     $\xrightarrow{\beta}$ rm_fst__(I,L,[],C+1).
rm_fst__(_,   [],A,C) $\xrightarrow{\gamma}$ {A,C+1};
rm_fst__(I,[I|L],A,C) $\xrightarrow{\delta}$ join(A,L,C+1);
rm_fst__(I,[J|L],A,C) $\xrightarrow{\epsilon}$ aux(I,L,join(A,[J],C+1)).
aux(I,L,{P,C})        $\xrightarrow{\zeta}$ rm_fst__(I,L,P,C).
```

This concludes the transformation.

What would have happened if we had started with a definition in tail form? Let us try with `rm_lst_tf/2`, as found on page 105:

```
rm_lst_tf(I,L)      -> rev_join(L,[],I).
rev_join(   [],Q,I) -> rm_fst(I,Q,[]);
rev_join([J|P],Q,I) -> rev_join(P,[J|Q],I).
rm_fst(_,   [],A)   -> A;
rm_fst(I,[I|Q],A)   -> rev_join(Q,A);
rm_fst(I,[J|Q],A)   -> rm_fst(I,Q,[J|A]).
```

We add a counter and a new clause to set its initial value. We increment it where the current clause corresponds to another one in the original definition. We must not forget to pair value and counter in the bodies without function calls (because the definition is in tail form, these bodies are the last expressions to be evaluated before the programs terminates). We get

```
rm_lst_tf(I,L)        -> rm_lst_tf(I,L,0).
rm_lst_tf(I,L,C)      -> rev_join(L,[],I,C+1).
rev_join(   [],Q,I,C) -> rm_fst(I,Q,[],C+1).
rev_join([J|P],Q,I,C) -> rev_join(P,[J|Q],I,C+1).
rm_fst(_,   [],A,C)   -> {A,C+1}
rm_fst(I,[I|Q],A,C)   -> rev_join(Q,A,C+1);
rm_fst(I,[J|Q],A,C)   -> rm_fst(I,Q,[J|A],C+1).
```

It becomes apparent now that adding an integer to count the number of function calls to a definition in tail form is very simple: just increment the counter everywhere. This would be the way to go if we planned to write a program adding these counters automatically: first make an automatic transformation into tail form and then add the counter in this simple manner.

Into tail form. Our definition of `flat/1` with lifting, on page 121,

```
flat(       []) -> [];
flat(   [[]|L]) -> flat(L);
flat([[I|M]|L]) -> flat([I,M|L]);
flat(    [I|L]) -> [I|flat(L)].
```

is almost in tail form: only the last clause has a call with a non-empty control context. Following the strategy proposed up to now, this means that, by adding an accumulator, this definition can be transformed into an equivalent one in tail form. The purpose of this accumulator is to store the variables which occur in the control contexts, so these can be rebuilt and computed after the current function is over.

Let us add a list accumulator `A`, unchanged in every clause, and add a new `flat_tf/1` definition calling the new `flat/2` with the initial value of the accumulator set to the empty list:

```
flat_tf(L)          $\xrightarrow{\alpha}$ flat(L,[]).
flat(       [],A) $\xrightarrow{\beta}$ [];                         % A unused yet
flat(   [[]|L],A) $\xrightarrow{\gamma}$ flat(L,A);
flat([[I|M]|L],A) $\xrightarrow{\delta}$ flat([I,M|L],A);
flat(    [I|L],A) $\xrightarrow{\epsilon}$ [I|flat(L,A)].
```

Now we must accumulate a value at each call which is not in tail form (here, clause ϵ), and use the contents of the accumulator in all clauses where there is no call (here, clause α). The technique consists in accumulating in clause ϵ the values occurring in the control context, which is `[I|␣]`; in other words: we push `I` onto `A`:

```
flat_tf(L)          $\xrightarrow{\alpha}$ flat(L,[]).
flat(       [],A) $\xrightarrow{\beta}$ [];                         % A unused yet
flat(   [[]|L],A) $\xrightarrow{\gamma}$ flat(L,A);
flat([[I|M]|L],A) $\xrightarrow{\delta}$ flat([I,M|L],A);
flat(    [I|L],A) $\xrightarrow{\epsilon}$ flat(L,[I|A]).
```

When the input is fully consummated, in clause β, the accumulator contains all the non-list items (all the `I`s from clause ϵ) in the reverse order of the original list; therefore, they need to be reversed. That is to say:

```
flat_tf(L)          $\xrightarrow{\alpha}$ flat(L,[]).
flat(       [],A) $\xrightarrow{\beta}$ rev(A);
flat(   [[]|L],A) $\xrightarrow{\gamma}$ flat(L,A);
flat([[I|M]|L],A) $\xrightarrow{\delta}$ flat([I,M|L],A);
flat(    [I|L],A) $\xrightarrow{\epsilon}$ flat(L,[I|A]).
```

What about `sflat/1` on page 118?

```
sflat(       []) -> [];
sflat(   [[]|L]) -> sflat(L);
sflat([[I|M]|L]) -> join(sflat([I|M]),sflat(L));
sflat(    [I|L]) -> [I|sflat(L)].
```

That definition has the peculiarity that some of its clauses contain two or more calls—let us not forget that a call being recursive or not has nothing to do, in general, with being in tail form or not. Let us start by adding the extra accumulative parameter to `sflat/1` and initialise it with the empty list:

```
sflat_tf(L)          $\xrightarrow{\alpha}$ sflat(L,[]).                   % Added
sflat(       [],A) $\xrightarrow{\gamma}$ [];                          % A unused yet
sflat(   [[]|L],A) $\xrightarrow{\delta}$ sflat(L,A);
sflat([[I|M]|L],A) $\xrightarrow{\epsilon}$ join(sflat([I|M],A),sflat(L,A));
sflat(    [I|L],A) $\xrightarrow{\zeta}$ [I|sflat(L,A)].
join(   [],Q)      $\xrightarrow{\eta}$ Q;
join([I|P],Q)      $\xrightarrow{\theta}$ [I|join(P,Q)].
```

Let us decide that, in clause ϵ, the first call to be rewritten is the leftmost recursive call `sflat([I|M],A)`, whose control context is thus `join(␣,sflat(L,A))`. Therefore, in clause ϵ, let us save `L` in `A` so we can reconstruct the control context in the body of γ, where the current list to process is empty and thus lists saved in the accumulator allow us to resume the flattening:

```
sflat_tf(L)              $\xrightarrow{\alpha}$ sflat(L,[]).
sflat(       [],[L|A]) $\xrightarrow{\gamma}$ join([],sflat(L,A));          % Used
sflat(   [[]|L],    A) $\xrightarrow{\delta}$ sflat(L,A);
sflat([[I|M]|L],    A) $\xrightarrow{\epsilon}$ sflat([I|M],[L|A]);           % Saved
sflat(    [I|L],    A) $\xrightarrow{\zeta}$ [I|sflat(L,A)].
join(   [],Q)           $\xrightarrow{\eta}$ Q;
join([I|P],Q)           $\xrightarrow{\theta}$ [I|join(P,Q)].
```

But a clause is now missing: what if the accumulator is empty? Therefore, a clause β must be added before clause γ to cater this situation:

```
sflat_tf(L)              $\xrightarrow{\alpha}$ sflat(L,[]).
sflat(       [],   []) $\xrightarrow{\beta}$ [];
sflat(       [],[L|A]) $\xrightarrow{\gamma}$ join([],sflat(L,A));
sflat(   [[]|L],    A) $\xrightarrow{\delta}$ sflat(L,A);
sflat([[I|M]|L],    A) $\xrightarrow{\epsilon}$ sflat([I|M],[L|A]);
sflat(    [I|L],    A) $\xrightarrow{\zeta}$ [I|sflat(L,A)].
join(   [],Q)           $\xrightarrow{\eta}$ Q;
join([I|P],Q)           $\xrightarrow{\theta}$ [I|join(P,Q)].
```

We can simplify the body of clause γ because of the algebraic identity

$$\texttt{join([],}L\texttt{)} \equiv L,$$

for all lists L and the definition of `join/2` becomes useless altogether.

```
sflat_tf(L)              $\xrightarrow{\alpha}$ sflat(L,[]).
sflat(       [],   []) $\xrightarrow{\beta}$ [];
sflat(       [],[L|A]) $\xrightarrow{\gamma}$ sflat(L,A);                   % Simplified
sflat(   [[]|L],    A) $\xrightarrow{\delta}$ sflat(L,A);
sflat([[I|M]|L],    A) $\xrightarrow{\epsilon}$ sflat([I|M],[L|A]);
sflat(    [I|L],    A) $\xrightarrow{\zeta}$ [I|sflat(L,A)].
```

Clause ζ is not in tail form. We cannot just push `I` onto the accumulator

```
sflat(    [I|L],    A) $\xrightarrow{\zeta}$ sflat(L,[I|A]).                 % Wrong
```

because the latter contains lists to be flattened later (see clause ϵ) and

`I` is not a list—this modification would lead to a match failure just after clause γ matches, because all heads only match lists. What can we do? Perhaps the first idea which comes to the mind is to add another accumulator to hold the non-list items, like `I`. Basically, this is exactly the same method as before, except it applies to another accumulator, say `B`. Let us first add `B` everywhere and initialise it with `[]`:

```
sflat_tf(L)                  $\xrightarrow{\alpha}$ sflat(L,[],[]).
sflat(       [],   [],B) $\xrightarrow{\beta}$ [];                    % B unused yet
sflat(       [],[L|A],B) $\xrightarrow{\gamma}$ sflat(L,A,B);
sflat(   [[]|L],     A,B) $\xrightarrow{\delta}$ sflat(L,A,B);
sflat([[I|M]|L],     A,B) $\xrightarrow{\epsilon}$ sflat([I|M],[L|A],B);
sflat(    [I|L],     A,B) $\xrightarrow{\zeta}$ [I|sflat(L,A,B)].
```

Now we can save the variables of the control context in clause ζ in `B` and erase the control context in question. In clause β, we know that `B` contains all the non-list items in reversed order, so we must reverse `B`. Since clause β contained no further calls, this is the end.

```
sflat_tf(L)                  $\xrightarrow{\alpha}$ sflat(L,[],[]).
sflat(       [],   [],B) $\xrightarrow{\beta}$ rev(B);
sflat(       [],[L|A],B) $\xrightarrow{\gamma}$ sflat(L,A,B);
sflat(   [[]|L],     A,B) $\xrightarrow{\delta}$ sflat(L,A,B);
sflat([[I|M]|L],     A,B) $\xrightarrow{\epsilon}$ sflat([I|M],[L|A],B);
sflat(    [I|L],     A,B) $\xrightarrow{\zeta}$ sflat(L,A,[I|B]).
```

Further examination can lead to a simpler program, where the heads do not only match embedded lists:

```
sflat_tf(L)         -> sflat(L,[],[]).
sflat(   [],[],B) -> rev(B);
sflat(   [], A,B) -> sflat(A,   [],    B);
sflat(  [I], A,B) -> sflat(I,    A,    B);        % Optimisation
sflat([I|L], A,B) -> sflat(I,[L|A],    B);
sflat(    I, A,B) -> sflat(A,   [],[I|B]).
```

The shortcoming of this approach is that it requires many accumulators in general and, in this particular example, it is too *ad hoc*, as it makes use of algebraic identities to simplify the definitions and the control context consisting of a push is reconstructed as a call to `rev/1`—which may not seem obvious.

Instead of adding a supplementary accumulator to solve our problem, we can stick to only one but make sure that values in it are distinguished according to their origin, so a value from a given control context is not confused with a value from another control context. This was previously implemented by using different accumulators for different context values. The way of achieving this with only one accumulator consists in putting in a tuple the values of a given control context together with an atom, which plays the role of a tag identifying the original expression containing the call. Let us backtrack to

```
sflat_tf(L)        -α-> sflat(L,[]).
sflat(       [],A) -γ-> [];                           % A unused yet
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ϵ-> join(sflat([I|M],A),sflat(L,A));
sflat(    [I|L],A) -ζ-> [I|sflat(L,A)].
join(   [],Q)      -η-> Q;
join([I|P],Q)      -θ-> [I|join(P,Q)].
```

Let us modify clause ϵ by choosing, as before, `sflat([I|M])` as the first call to be rewritten. We choose the atom `k1` to represent that call and we pair it with the sole value of its control context, that is, `L`. We remove the control context `join(␣,sflat(L,A))` and, in the remaining call, we push `{k1,L}` onto the accumulator `A`:

```
sflat_tf(L)        -α-> sflat(L,[]).
sflat(       [],A) -γ-> [];                           % A unused yet
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ϵ-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A) -ζ-> [I|sflat(L,A)].
join(   [],Q)      -η-> Q;
join([I|P],Q)      -θ-> [I|join(P,Q)].
```

The key point is that `k1` must not be pushed in this accumulator anywhere else in the program, because it must denote unambiguously the call in clause ϵ. Of course, this program is not correct anymore, as the erased control context must be reconstructed somewhere else and applied to the value of the call `sflat([I|M],[{k1,L}|A])`. The accumulator `A` represents, as before, the values of the control contexts. Where should we extract its contents? Clause γ does not make any use of `A` and this is our cue. It means that, at that point, there are no more lists to be flattened, beyond the trivial empty list, so this is the right moment to wonder if there is some work left to be done, that is, examine the contents of `A`. In order to implement this task, a dedicated function should be created, say `appk/2`, so that `appk(`V`,`A`)` will compute whatever remains to be done with what is found in the accumulator A, the value V being a partial result, that is, the result up to this point. Hence, if there is nothing left to do, that is, if A is empty, then `appk(`V`,`A`)` rewrites into V and this is it. In other words:

`appk(V,[{k1,L}|A])` $\xrightarrow{\kappa}$ $\boxed{}$;

`appk(V,         [])` $\xrightarrow{\iota}$ `V.`   *% The end*

The empty box must be filled with the reconstruction of the control context which was erased at the point where `k1` was saved in the accumulator. The control context in question was `join(␣,sflat(L,A))`, in clause ϵ, so we have now

`appk(V,[{k1,L}|A])` $\xrightarrow{\kappa}$ `join(□,sflat(L,A));`
`appk(V,         [])` $\xrightarrow{\iota}$ `V.`

The remaining empty box is meant to be filled with the result of the function call `sflat([I|M],[{k1,L}|A])`. To make this happen, two conditions must be fulfilled. Firstly, the accumulator in the head of `appk/2` must be the same as at the moment of the call, that is, it must be matched by `[{k1,L}|A]`. In theory, we should prove that the two occurrences of `A` indeed denote the same value, but this would lead us astray. Finally, we need to make sure that when a call to `sflat/2` is over, a call to `appk/2` is made with the result. When the whole transformation into tail form will be completed, no control context will be found anymore (by definition), so all calls to `sflat/2` will end in clauses whose bodies do not contain any further call to be processed. A quick examination of the clauses reveals that clause γ is the only clause of concern and that `A` was unused yet. So let us replace the body of this clause with a call to `appk/2`, whose first argument is the result of the current call to `sflat/2`, that is, the current body, and whose second argument is the accumulator which may contain more information about control contexts to be rebuilt and applied. We have

`sflat(        [],A)` $\xrightarrow{\gamma}$ **`appk([],A);`**

Now we understand that `V` in clause κ is the value of the function call `sflat([I|M],[{k1,L}|A])`, so we can proceed by plugging `V` into the place-holder in clause κ:

`appk(V,[{k1,L}|A])` $\xrightarrow{\kappa}$ `join(`**`V`**`,sflat(L,A));`

A glance is enough to realise that clause κ is not in tail form. Therefore, let us repeat the same method. The first call that must be rewritten is `sflat(L,A)`, whose control context is `join(V,␣)`. Let us associate the variable `V` in the latter with a new atom `k2` and push the two of them onto the accumulator:

`appk(V,[{k1,L}|A])` $\xrightarrow{\kappa}$ `sflat(L,`**`[{k2,V}|`**`A]);`

We need a new clause for `appk/2` which handles the corresponding case, that is, when the value of the call has been found and the control context has to be reconstructed and resumed:

`sflat_tf(L)          ` $\xrightarrow{\alpha}$ `sflat(L,[]).`

```
sflat(       [],A) -γ-> appk([],A);
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ϵ-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A) -ζ-> [I|sflat(L,A)].
join(   [],Q)      -η-> Q;
join([I|P],Q)      -θ-> [I|join(P,Q)].
appk(V,[{k2,W}|A]) -λ-> join(W,V);               % A unused yet
appk(V,[{k1,L}|A]) -κ-> sflat(L,[{k2,V}|A]);
appk(V,        []) -ι-> V.
```

Notice how, in clause λ, we renamed `V` (in the accumulator) into `W`, so as to avoid a clash with the first argument of `appk/2`. Also, why is it `join(W,V)` and not `join(V,W)`? The reason is found by recollecting that `W` denotes the value of the call `sflat([I|M])` (in the original definition), whereas `V` represents the value of `sflat(L)` (in the original definition). Nothing is done yet with the rest of the accumulator `A`, which entails that we must pass it to `join/2`, just like the other functions:

```
sflat_tf(L)        -α-> sflat(L,[]).
sflat(       [],A) -γ-> appk([],A);
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ϵ-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A) -ζ-> [I|sflat(L,A)].
join(   [],Q,A)    -η-> Q;                       % A unused yet
join([I|P],Q,A)    -θ-> [I|join(P,Q,A)].
appk(V,[{k2,W}|A]) -λ-> join(W,V,A);                  % Passed A
appk(V,[{k1,L}|A]) -κ-> sflat(L,[{k2,V}|A]);
appk(V,        []) -ι-> V.
```

After clause ϵ, the first clause not being in tail form is clause ζ. Let us pair the variable `I` of the control context `[I|␣]` with a new atom `k3`, and let us save the pair into the accumulator, while reconstructing the erased control context in a new clause μ of `appk/2`:

```
sflat_tf(L)        -α-> sflat(L,[]).
sflat(       [],A) -γ-> appk([],A);
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ϵ-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A) -ζ-> sflat(L,[{k3,I}|A]).          % I saved
join(   [],Q,A)    -η-> Q;                       % A unused yet
join([I|P],Q,A)    -θ-> [I|join(P,Q,A)].
appk(V,[{k3,I}|A]) -μ-> [I|V];                   % A unused yet
appk(V,[{k2,W}|A]) -λ-> join(W,V,A);
appk(V,[{k1,L}|A]) -κ→ sflat(L,[{k2,V}|A]);
appk(V,        []) -ι→ V.
```

Something interesting happens here: the brand-new body of clause μ makes no use of the remaining accumulator `A`. We encountered the exact same situation with γ: a body containing no further calls. In this case, we need to check whether there is more work to be done with the data saved earlier in `A`. This is the very aim of `appk/2`, therefore a call to it must be set within the body of clause μ:

```
sflat_tf(L)         -α→ sflat(L,[]).
sflat(        [],A) -γ→ appk([],A);
sflat(    [[]|L],A) -δ→ sflat(L,A);
sflat([[I|M]|L],A) -ϵ→ sflat([I|M],[{k1,L}|A]);
sflat(     [I|L],A) -ζ→ sflat(L,[{k3,I}|A]).
join(   [],Q,A)     -η→ Q;                           % A unused yet
join([I|P],Q,A)     -θ→ [I|join(P,Q,A)].
appk(V,[{k3,I}|A]) -μ→ appk([I|V],A);
appk(V,[{k2,W}|A]) -λ→ join(W,V,A);
appk(V,[{k1,L}|A]) -κ→ sflat(L,[{k2,V}|A]);
appk(V,        []) -ι→ V.
```

The next clause to be considered is clause η, because its body contains no calls, so it requires now a call to `appk/2` with the body, which is the result of the current call to `sflat/2`, and the accumulator, that is, `A`:

```
sflat_tf(L)         -α→ sflat(L,[]).
sflat(        [],A) -γ→ appk([],A);
sflat(    [[]|L],A) -δ→ sflat(L,A);
sflat([[I|M]|L],A) -ϵ→ sflat([I|M],[{k1,L}|A]);
sflat(     [I|L],A) -ζ→ sflat(L,[{k3,I}|A]).
join(   [],Q,A)     -η→ appk(Q,A);
join([I|P],Q,A)     -θ→ [I|join(P,Q,A)].
appk(V,[{k3,I}|A]) -μ→ appk([I|V],A);
appk(V,[{k2,W}|A]) -λ→ join(W,V,A);
appk(V,[{k1,L}|A]) -κ→ sflat(L,[{k2,V}|A]);
appk(V,        []) -ι→ V.
```

Last but not least, clause θ must be fixed as we did for the other clauses not in tail form. Let us pick a new atom, say, `k4`, and tuple it with the sole variable `I` of the control context `[I|␣]` and push the resulting pair onto the accumulator `A`. Dually, we need to add a clause ν to `appk/2` to catch this case, rebuild the erased control context and apply it to the result of the current call to `sflat/2`, that is, its first argument:

```
sflat_tf(L)         -α→ sflat(L,[]).
```

```
sflat(       [],A) -γ-> appk([],A);
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ε-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A) -ζ-> sflat(L,[{k3,I}|A]).
join(   [],Q,A)    -η-> appk(Q,A);
join([I|P],Q,A)    -θ-> join(P,Q,[I|A]).
appk(V,[{k4,I}|A]) -ν-> [I|V];                     % A unused yet
appk(V,[{k3,I}|A]) -μ-> appk([I|V],A);
appk(V,[{k2,W}|A]) -λ-> join(W,V,A);
appk(V,[{k1,L}|A]) -κ-> sflat(L,[{k2,V}|A]);
appk(V,        []) -ι-> V.
```

The newly created clause contains a body which contains no calls, so we must send it to `appk/2` together with the rest of the accumulator, in order to process any pending control contexts:

```
sflat_tf(L)         -α-> sflat(L,[]).
sflat(       [],A) -γ-> appk([],A);
sflat(   [[]|L],A) -δ-> sflat(L,A);
sflat([[I|M]|L],A) -ε-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A) -ζ-> sflat(L,[{k3,I}|A]).
join(   [],Q,A)    -η-> appk(Q,A);
join([I|P],Q,A)    -θ-> join(P,Q,[{k4,I}|A]).
appk(V,[{k4,I}|A]) -ν-> appk([I|V],A);
appk(V,[{k3,I}|A]) -μ-> appk([I|V],A);
appk(V,[{k2,W}|A]) -λ-> join(W,V,A);
appk(V,[{k1,L}|A]) -κ-> sflat(L,[{k2,V}|A]);
appk(V,        []) -ι-> V.
```

The transformation is now finished. It is correct in the sense that the resulting program is equivalent to the original one, that is, `sflat/1` and `sflat_tf/1` compute the same values from the same inputs, and all the clauses are in tail form. It is also complete in the sense that any definition can be transformed. Notice how we did not make any use of algebraic identities to simplify and speed up `sflat_tf/1`, contrary to the previous *ad hoc* transformation. As announced, the main interest of this method lies in its uniformity and must not be expected to generate programs which are faster than the originals.

It is possible, upon close examination, to shorten a bit the definition of `appk/2`. Indeed, clauses ν and μ are identical, if not the presence of a different tag, `k4` versus `k3`. Let us fuse them into a single clause and use a new atom `k34` instead of every occurrence of `k3` and `k4`.

```
sflat_tf(L)         -α-> sflat(L,[]).
```

```
sflat(       [],A)  -γ→ appk([],A);
sflat(   [[]|L],A)  -δ→ sflat(L,A);
sflat([[I|M]|L],A)  -ε→ sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A)  -ζ→ sflat(L,[{k34,I}|A]).
join(   [],Q,A)     -η→ appk(Q,A);
join([I|P],Q,A)     -θ→ join(P,Q,[{k34,I}|A]).
appk(V,[{k34,I}|A]) -μ→ appk([I|V],A);
appk(V,[{k2,W}|A])  -λ→ join(W,V,A);
appk(V,[{k1,L}|A])  -κ→ sflat(L,[{k2,V}|A]);
appk(V,        [])  -ι→ V.
```

Let us make a short digression and transform `sflat_tf/1` further so that `sflat_tf(`L`)` is rewritten into a pair made of the value of `sflat(`L`)` and its delay. As we saw on page 137, because the definition is initially in tail form, we just have to add a counter and increment it where the clause corresponds to a clause in the original definition, else the counter is left unchanged. We also have to add a clause to set the first value of the counter. Let us recall first the original definition of `sflat/1` on page 118 (we rename the arrows here to ease the forthcoming steps):

```
sflat(       []) -γ→ [];
sflat(   [[]|L]) -δ→ sflat(L);
sflat([[I|M]|L]) -ε→ join(sflat([I|M]),sflat(L));
sflat(    [I|L]) -ζ→ [I|sflat(L)].
join(   [],Q)    -η→ Q;
join([I|P],Q)    -θ→ [I|join(P,Q)].
```

Then, let us identify and name identically in the tail form version `sflat_tf/1` the clauses that have their counterpart in the definition of `sflat/1`:

```
sflat_tf(L)          → sflat(L,[]).
sflat(       [],A)  -γ→ appk([],A);
sflat(   [[]|L],A)  -δ→ sflat(L,A);
sflat([[I|M]|L],A)  -ε→ sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A)  -ζ→ sflat(L,[{k34,I}|A]).
join(   [],Q,A)     -η→ appk(Q,A);
join([I|P],Q,A)     -θ→ join(P,Q,[{k34,I}|A]).
appk(V,[{k34,I}|A])  → appk([I|V],A);
appk(V,[{k2,W}|A])   → join(W,V,A);
appk(V,[{k1,L}|A])   → sflat(L,[{k2,V}|A]);
appk(V,        [])   → V.
```

Now we can add the counters and increment them only on the distinguished clauses:

```
sflat_tf(L)            →  sflat_tf(L,0).                     % New
sflat_tf(L,C)          →  sflat(L,[],C).
sflat(        [],A,C)  γ→ appk([],A,C+1);
sflat(    [[]|L],A,C)  δ→ sflat(L,A,C+1);
sflat([[I|M]|L],A,C)   ϵ→ sflat([I|M],[{k1,L}|A],C+1);
sflat(     [I|L],A,C)  ζ→ sflat(L,[{k34,I}|A],C+1).
join(    [],Q,A,C)     η→ appk(Q,A,C+1);
join([I|P],Q,A,C)      θ→ join(P,Q,[{k34,I}|A],C+1).
appk(V,[{k34,I}|A],C)  →  appk([I|V],A,C);
appk(V,[{k2,W}|A],C)   →  join(W,V,A,C);
appk(V,[{k1,L}|A],C)   →  sflat(L,[{k2,V}|A],C);
appk(V,          [],C) →  {V,C}.
```

Note how the last clause of `appk/3` always implements the last rewrite, so this is the only place where we must care to create the expected pair.

Drawing from our practical understanding of the new, systematic transformation, we can try to summarise it as follows.

1. Consider all the definitions involved, that is, the one of immediate concern, but also all which it depends upon;
2. add a list accumulator to all these definitions and add a definition setting the empty list as the initial value of the accumulator;
3. for each body made of a call in tail form, just pass the accumulator unchanged;
4. replace each body containing no call by a call to a new function `appk/2`, with the body expression and the accumulator unchanged;
5. for each body not in tail form, including those of `appk/2`,
   (a) identify or choose the first possible call to be computed;
   (b) select all the values and variables in the control context which are parameters, except the accumulator, and group them in a tuple, together with a unique atom;
   (c) replace the body in question with the call to be done first and pass to it the accumulator on top of which the tuple of the previous step has been pushed;
   (d) create a clause for `appk/2` matching this case, whose body is the previously mentioned control context;
   (e) replace the place-holder ␣ in the control context by the first argument of `appk/2` and make sure that there is no clash of variables;
6. add the clause `appk(V,[]) -> V` to `appk/2`.

This algorithm is said to be *global*, insofar as *all* the steps must be achieved before a program equivalent to the original input is reached, because intermediary steps may not lead to correct definitions. It is possible to dynamically rearrange the order in which some steps are applied so the algorithm becomes *incremental*, but it is probably not worth the complication.

Let us apply the same methodological steps to another difficult definition like `fib/1`, as given on page 16:

```
fib(0)            $\xrightarrow{\beta}$ 1;
fib(1)            $\xrightarrow{\gamma}$ 1;
fib(N) when N > 1 $\xrightarrow{\delta}$ fib(N-1) + fib(N-2).
```

The steps are as follows.

1. This definition is self-contained.
2. Let us rename `fib/1` into `fib/2`, then add a list accumulator to it so it becomes `fib/2`, next create a clause α defining `fib_tf/1` as a single call to `fib/2` where the initial value of the accumulator is the empty list:

```
fib_tf(N)           $\xrightarrow{\alpha}$ fib(N,[]).                % New
fib(0,A)            $\xrightarrow{\beta}$ 1;
fib(1,A)            $\xrightarrow{\gamma}$ 1;
fib(N,A) when N > 1 $\xrightarrow{\delta}$ fib(N-1,A) + fib(N-2,A).
```

3. There is no body in tail form which contains a call.
4. Clauses β and γ are in tail form and contain no call, so we must replace the bodies with a call to function `appk/2`, whose first argument is the original body (here, both are the value `1`) and the second argument is the accumulator unchanged:

```
fib_tf(N)           $\xrightarrow{\alpha}$ fib(N,[]).
fib(0,A)            $\xrightarrow{\beta}$ appk(1,A);
fib(1,A)            $\xrightarrow{\gamma}$ appk(1,A);
fib(N,A) when N > 1 $\xrightarrow{\delta}$ fib(N-1,A) + fib(N-2,A).
```

5. Clause δ is not in tail form and contains two calls, so we must choose which one we want to compute first. Let us arbitrarily choose the rightmost call, that is, `fib(N-2,A)`. Therefore, its control context is `fib(N-1,A) + ␣`. The values in the control context, excluding the accumulator, are reduced to the sole value of `N`. Let us create a unique atom identifying this call, `k1`, and form the pair `{k1,N}`. Then, let us replace the entire body of clause δ with `fib(N-2,[{k1,N}|A])`. Next, let us create a clause for `appk/2` matching this tuple. Its body is the control context we just removed from the body of clause δ. In it, let us fill the hole ␣ with

the first parameter.

```
fib_tf(N)            $\xrightarrow{\alpha}$ fib(N,[]).
fib(0,A)             $\xrightarrow{\beta}$ appk(1,A);
fib(1,A)             $\xrightarrow{\gamma}$ appk(1,A);
fib(N,A) when N > 1  $\xrightarrow{\delta}$ fib(N-2,[{k1,N}|A]).
appk(V,[{k1,N}|A])   $\xrightarrow{\epsilon}$ fib(N-1,A) + V.
```

The body of the clause handling `k1` is not in tail form, as it contains a function call not located at the root of the abstract syntax tree. The control context of this call is `␣ + V` and all the values it contains are limited to the one denoted by the variable `V`. Let us generate a new unique atom `k2` and pair it with `V`. We then replace the body of clause ϵ with the call to be computed first and we pass to it the accumulator `A` on top of which the pair has been pushed. We make a new clause of `appk/2` matching this case and in its body we put the control context we just mentioned. We substitute the first parameter to the place-holder `␣`. We have

```
appk(V,[{k2,W}|A]) $\xrightarrow{\zeta}$ V + W;
appk(V,[{k1,N}|A]) $\xrightarrow{\epsilon}$ fib(N-1,[{k2,V}|A]).
```

Note that we carefully renamed the variable `V` in the accumulator into `W` in order to avoid a clash with the first parameter `V`. This new body `V+W` is in tail form and contains no further function calls, so it must be embedded into a recursive call because the accumulator `A` may not be empty—so further calls may be waiting. We pass to the call the remaining accumulator, that is, `A`. Finally, all the clauses are in tail form:

```
fib_tf(N)            $\xrightarrow{\alpha}$ fib(N,[]).
fib(0,A)             $\xrightarrow{\beta}$ appk(1,A);
fib(1,A)             $\xrightarrow{\gamma}$ appk(1,A);
fib(N,A) when N > 1  $\xrightarrow{\delta}$ fib(N-2,[{k1,N}|A]).
appk(V,[{k2,W}|A])   $\xrightarrow{\zeta}$ appk(V+W,A);
appk(V,[{k1,N}|A])   $\xrightarrow{\epsilon}$ fib(N-1,[{k2,V}|A]).
```

6. We must make sure to add a clause to match the case of the empty accumulator and rewrite to the first parameter:

```
fib_tf(N)            $\xrightarrow{\alpha}$ fib(N,[]).
fib(0,A)             $\xrightarrow{\beta}$ appk(1,A);
fib(1,A)             $\xrightarrow{\gamma}$ appk(1,A);
fib(N,A) when N > 1  $\xrightarrow{\delta}$ fib(N-2,[{k1,N}|A]).
appk(V,        [])   $\xrightarrow{\eta}$ V;                   % Do not forget!
appk(V,[{k2,W}|A])   $\xrightarrow{\zeta}$ appk(V+W,A);
```

appk(V,[{k1,N}|A]) $\xrightarrow{\epsilon}$ fib(N-1,[{k2,V}|A]).

Let us apply now our general method to flat/1. Let us pick up here:

```
flat_tf(L)        -> flat(L,[]).
flat(       [],A) -> [];                          % A unused yet
flat(   [[]|L],A) -> flat(L,A);
flat([[I|M]|L],A) -> flat([I,M|L],A);
flat(    [I|L],A) -> [I|flat(L,A)].
```

The only body containing no calls is in the first clause of flat/2, so it must be applied to a call to appk/2, together with the accumulator. Only the last body is not in tail form. The only call to be performed has the control context [I|␣], whose only values are reduced to the sole I. So we generate a unique atom k1 and we pair it with I. We replace the body not in tail form with the call to which we pass the accumulator on top of which the pair has been pushed. We consequently create a clause for appk/2 matching this case. Its body is the just erased control context. The hole ␣ is filled with the first parameter:

```
flat_tf(L)         -> flat(L,[]).
flat(       [],A)  -> appk([],A);
flat(   [[]|L],A)  -> flat(L,A);
flat([[I|M]|L],A)  -> flat([I,M|L],A);
flat(    [I|L],A)  -> flat(L,[{k1,I}|A]).
appk(V,[{k1,I}|A]) -> [I|V].
```

Since the body of the newly created clause of appk/2 is a value, it has to be wrapped into a recursive call because the accumulator A may not be empty, so perhaps some more calls have to be computed:

```
flat_tf(L)         -> flat(L,[]).
flat(       [],A)  -> appk([],A);
flat(   [[]|L],A)  -> flat(L,A);
flat([[I|M]|L],A)  -> flat([I,M|L],A);
flat(    [I|L],A)  -> flat(L,[{k1,I}|A]).
appk(V,[{k1,I}|A]) -> appk([I|V],A).
```

Finally, the definition of appk/2 must be completed by a clause corresponding to the case when the accumulator is empty and its body simply returns the first argument, which is, by design, the result:

```
flat_tf(L)         -> flat(L,[]).
flat(       [],A)  -> appk([],A);
flat(   [[]|L],A)  -> flat(L,A);
flat([[I|M]|L],A)  -> flat([I,M|L],A);
flat(    [I|L],A)  -> flat(L,[{k1,I}|A]).
```

```
appk(V,        []) -> V;
appk(V,[{k1,I}|A]) -> appk([I|V],A).
```

If we compare this version with

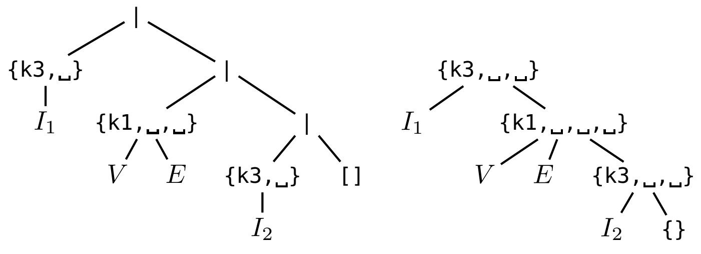
```
flat_tf(L)        -> flat(L,[]).
flat(       [],A) -> rev(A);
flat(   [[]|L],A) -> flat(L,A);
flat([[I|M]|L],A) -> flat([I,M|L],A);
flat(    [I|L],A) -> flat(L,[I|A]).
```

we understand that the latter can be derived from the former if the pair `{k1,I}` is replaced by `I`. This is possible because it is the only atom which was generated. The definition of `appk/2` then is equivalent to `rev_join/2`, on page 60:

```
rev(L)            -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

The philosophy underlying our general method to transform a given group of definitions into an equivalent in tail form consists in adding a parameter which is a list accumulating the values of the different control contexts and creating a function (`appk/2`) to reconstruct these when the call they contained is over. These rebuilt control contexts are in turn transformed into tail form until all the clauses are in tail form. As a result, the number of clauses is larger than in the original source and the algorithm is obscured because of all the administrative work about the accumulator. Even though the call stack is managed by the run-time system much more efficiently than the heap, the often greater number of function calls makes it hard to guarantee a speedup every time a tail form is used. In order to save time and efforts, it is wise to consider tail forms useful *a posteriori*, when we run afoul of the maximum stack size because, except if very large inputs are, from the design phase, likely.

Lighter encoding of linear accumulators. The accumulators used to transform definitions into tail form are, in their most general instance, lists of tuples. While using a list brings to the fore the very nature of the accumulator, it incurs a penalty in the size of the memory required because, in the abstract syntax trees, a push corresponds to a node, just as a tuple. *By nesting tuples in tuples, we can get rid of the list altogether.* For instance, instead of writing `[{k3,`I_1`},{k1,`V`,`E`},{k3,`I_2`}]`, we would write the nested tuples `{k3,`I_1`,{k1,`V`,`E`,{k3,`I_2`,{}}}}`. Both abstract syntax trees are easily compared in FIGURE 28 on page 152. The encoding of a list accumulator



(a) With a list of tuples (b) With nested tuples

FIGURE 28: Two implementations of the same linear accumulator

by means of tuples only supposes to add a component to each tuple, which holds what was the "next" tuple in the list. The memory saving consists in one edge for each initial tuple, plus all the push nodes, that is, if there were n tuples, we save n edges (often called *pointers* in imperative languages) and n nodes. This is a very significant amelioration. As an illustration, let us improve on the following code we derived earlier:

```
sflat_tf(L)         -α-> sflat(L,[]).
sflat(       [],A)  -γ-> appk([],A);
sflat(   [[]|L],A)  -δ-> sflat(L,A);
sflat([[I|M]|L],A)  -ε-> sflat([I|M],[{k1,L}|A]);
sflat(    [I|L],A)  -ζ-> sflat(L,[{k34,I}|A]).
join(   [],Q,A)     -η-> appk(Q,A);
join([I|P],Q,A)     -θ-> join(P,Q,[{k34,I}|A]).
appk(V,[{k34,I}|A]) -μ-> appk([I|V],A);
appk(V,[{k2,W}|A])  -λ-> join(W,V,A);
appk(V,[{k1,L}|A])  -κ-> sflat(L,[{k2,V}|A]);
appk(V,        [])  -ι-> V.
```

It results in the more economical

```
sflat_tf(L)         -α-> sflat(L,{}).
sflat(       [],A)  -γ-> appk([],A);
sflat(   [[]|L],A)  -δ-> sflat(L,A);
sflat([[I|M]|L],A)  -ε-> sflat([I|M],{k1,L,A});
sflat(    [I|L],A)  -ζ-> sflat(L,{k34,I,A}).
join(   [],Q,A)     -η-> appk(Q,A);
join([I|P],Q,A)     -θ-> join(P,Q,{k34,I,A}).
appk(V,{k34,I,A})   -μ-> appk([I|V],A);
appk(V,{k2,W,A})    -λ-> join(W,V,A);
appk(V,{k1,L,A})    -κ-> sflat(L,{k2,V,A});
appk(V,      {})    $\xrightarrow{\iota}$ V.
```

Improvements. Just to illustrate the point that improvements on a definition which is not in tail form are much more beneficial than a mere transformation to tail form, let us consider again the Fibonacci function:

```
fib(0)              -> 1;
fib(1)              -> 1;
fib(N) when N > 1 -> fib(N-1) + fib(N-2).
```

The equations defining the delay of this function are simply:

$$\mathcal{D}_0^{\texttt{fib}} := 1; \quad \mathcal{D}_1^{\texttt{fib}} := 1; \quad \mathcal{D}_n^{\texttt{fib}} := 1 + \mathcal{D}_{n-1}^{\texttt{fib}} + \mathcal{D}_{n-2}^{\texttt{fib}}, \text{ with } n > 1.$$

Adding 1 on both sides of the last equation and reordering the terms:

$$\mathcal{D}_n^{\texttt{fib}} + 1 = (\mathcal{D}_{n-1}^{\texttt{fib}} + 1) + (\mathcal{D}_{n-2}^{\texttt{fib}} + 1).$$

This gives us the idea to set $D_n := \mathcal{D}_n^{\texttt{fib}} + 1$, yielding, for $n > 1$,

$$D_0 = \mathcal{D}_0^{\texttt{fib}} + 1 = 2, \quad D_1 = \mathcal{D}_1^{\texttt{fib}} + 1 = 2, \quad D_n = D_{n-1} + D_{n-2}.$$

The recurrence is the same as the Fibonacci sequence (third clause of `fib/1`), except for D_0 and D_1 whose values are 2 instead of 1. In order to make it coincide with the values of `fib/1`, we need to set $F_n := D_n/2$:

$$\mathcal{D}_n^{\texttt{fib}} = 2 \cdot F_n - 1.$$

Now we have $F_0 = F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$, for all $n > 1$; importantly, F_n computes the same values as `fib/1`, that is, $F_n \equiv$ `fib`(n). Let us prove now by means of *complete induction* on $n > 0$ that

$$F_0 = 1; \quad F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n), \text{ where } \phi := \frac{1+\sqrt{5}}{2} \text{ and } \hat{\phi} := 1 - \phi.$$

First, let us verify that the formula works for the smallest value of n:

$$F_1 = \frac{1}{\sqrt{5}}(\phi - \hat{\phi}) := \frac{1}{\sqrt{5}}(\phi - (1 - \phi)) = 1.$$

Let us suppose now that the equation to establish is valid for all values ranging from 1 to n (this is the *complete induction hypothesis*) and let us prove that it holds for $n+1$. We have, $F_{n+1} := F_n + F_{n-1}$. We can use the complete induction hypothesis for the cases $n-1$ and n:

$$\begin{aligned}
F_{n+1} &= \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) + \frac{1}{\sqrt{5}}(\phi^{n-1} - \hat{\phi}^{n-1}) \\
&= \frac{1}{\sqrt{5}}((\phi^n + \phi^{n-1}) - (\hat{\phi}^n + \hat{\phi}^{n-1})) \\
&= \frac{1}{\sqrt{5}}(\phi^{n-1}(\phi + 1) - \hat{\phi}^{n-1}(\hat{\phi} + 1)).
\end{aligned}$$

The key is that ϕ and $\hat{\phi}$ are the roots of the polynomial equation $x^2 = x + 1$, therefore

$$F_{n+1} = \frac{1}{\sqrt{5}}(\phi^{n-1} \cdot \phi^2 - \hat{\phi}^{n-1} \cdot \hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}),$$

which was the statement to be proved. The complete induction principle then implies that the equation holds for all $n > 0$. Now that we derived a closed form for F_n, let us study its asymptotic behaviour. This is straightforward if we start by noticing that $\hat{\phi} < 1$. Therefore $\hat{\phi}^n \rightarrow 0$, as n gets large and, because $\phi > 1$, only remains, as $n \rightarrow \infty$,

$$F_n \sim \frac{1}{\sqrt{5}}\phi^n, \text{ implying } \mathcal{D}_n^{\texttt{fib}} \sim \frac{2}{\sqrt{5}}\phi^n.$$

That is, this delay is *exponential* and, because $\phi > 1$, it will always be greater than any polynomial delay, except perhaps for a finite number of some small values of n. In other words, this is hopelessly slow.

How can we improve this definition?

We must resist the temptation to transform it into tail form because being in tail form only benefits the call stack, not the delay in general. By looking back at the call tree of `fib(5)`, on page 18, we realise that some small subtrees are duplicated, like the ones rooted at `fib(2)` and, even larger ones, like `fib(3)`. Let us examine the leftmost branch, from the leaf to the root. It is made of the successive nodes `fib(1)`, `fib(2)`, `fib(3)`, `fib(4)` and `fib(5)`, that is, all the values of `fib(N)` for `N` ranging from `1` to `5`. Generalising this observation, we can say that the series $(\texttt{fib(N)})_{\texttt{N}}$ is entirely described, except `fib(0)`, by the leftmost branch in the call tree of `fib(N)`. Therefore, starting from the small tree

$$\begin{tikzcd}[column sep=tiny]
& \texttt{fact(2)} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{fact(1)} & & \texttt{fact(0)}
\end{tikzcd}$$

we can obtain the complete call tree for `fib(5)` by growing the tree from the root, whilst sharing some subtrees, that is, reusing them instead of recomputing them, so the call tree looks now like in FIGURE 29 on page 155 (technically, it is not a simple tree but a more general *Directed Acyclic Graph*, where the curved arrowed edges implement the reuse of subtrees. This graph representation leads us to think that if two successive Fibonacci numbers are kept at all times, we can achieve this maximal sharing. Let us denote by F_n the nth Fibonacci number in the series. Then each computational step is $(F_{n-1}, F_n) \rightarrow (F_n, F_{n+1}) := (F_n, F_n + F_{n-1})$. In other words, let f be the function such that $f(x, y) := (y, x + y)$, then $(F_n, F_{n+1}) = f(F_{n-1}, F_n)$ and

$$(F_n, F_{n+1}) = f(F_{n-1}, F_n) = f(f(F_{n-2}, F_{n-1})) = f^2(F_{n-2}, F_{n-1})$$

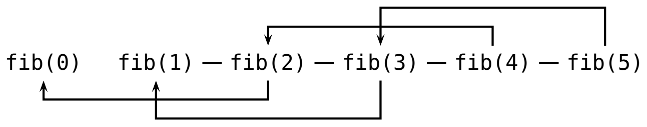


FIGURE 29: Call tree of `fib(5)` with maximal sharing

etc. till we reach $(F_n, F_{n+1}) = f^n(F_0, F_1) := f^n(1,1)$, for all $n \geqslant 0$. Let π_1 be the function such that $\pi_1(x,y) = x$, that is, it projects the first component of a pair, then $F_n = \pi_1 \circ f^n(1,1)$, for all $n \geqslant 0$. The iteration of f is easy to define by the recurrent equations

$$f^0(x,y) = (x,y),$$
$$f^n(x,y) = f^{n-1}(f(x,y)) := f^{n-1}(y, x+y), \qquad \text{where } n > 0.$$

The Erlang code is now straightforward:

```
fib_opt(N) -> pi1(f(N,{1,1})).
pi1({X,_}) -> X.
f(0,{X,Y}) -> {X,Y};
f(N,{X,Y}) -> f(N-1,{Y,X+Y}).
```

A tail form definition is extremely easy to obtain, without applying the general method:

```
fib_opt_tf(N) -> f(N,{1,1}).
f(0,{X,_})    -> X;                       % Projection done here
f(N,{X,Y})    -> f(N-1,{Y,X+Y}).
```

We deduce the delay $\mathcal{D}_n^{\texttt{fib\_opt\_tf}} = n + 2 \sim n$, as $n \to \infty$, which is thus asymptotically *linear*. This is a tremendous improvement over `fib/1` and, as an unexpected bonus, the definition is in tail form and is made of the same number of clauses as the original.

The general algorithm we presented in this section transforms all the definitions of the functions used by a given definition. Assuming that the size of the control stack is a real issue, is it possible not to transform all the functions involved? Consider again `rm_lst/2`, defined on page 97:

```
rm_lst(I,L) -> rev(rm_fst(I,rev(L))).
```

If we use the alternative definition `rm_fst_tf/2`, which is in tail form, instead of `rm_fst/2`, and, since `rev/1` is already in tail form, we reach

```
rm_lst(I,L) -> rev(rm_fst_tf(I,rev(L))).
```

where all the composed functions are in tail form. Of course, a function composition, like `rm_fst/2`, is not, by definition, in tail form, but it is not a problem. The size of control stack needed to compute the calls to `rm_lst/2` will be bounded by a small constant, because it is not recursive. That is why many good programmers only worry of having recursive functions in tail form and that function definitions in *tail form* are often called by reductionism *tail-recursive.*

Exercises. [See answers on page 377.]

1. Define a function `split/2` such that the call split(L,n), where L is a list and n an integer, is rewritten into a pair of lists, the first containing the first n items of L in the original order, the second containing the remaining items in the original order as well. If $n \leqslant 0$ or n is the length of L, then split(L,n) is undefined (if the latter case were allowed, there would be an ambiguity between two possible outcomes: {L,[]} and {[],L}). For example,

   ```
   split([a,b,c,d,e,f,g],5) ↠ {[a,b,c,d,e],[f,g]};
   split([a,b,c,d,e,f,g],1) ↠ {[a],[b,c,d,e,f,g]}.
   ```

   Provide a tail form as well. Discuss the delay and the possible best and worst cases.

2. Define a function `rot/2` such that the call rot(L,n), where L is a list and n an integer, is rewritten into the list containing the items of L such that the len(L) - n last items of L come first (in the original order), followed by the n first items of L (in the original order). This is usually called a *circular rotation.* For instance,

   ```
     rot([a,b,c,d,e,f,g],5) ↠ [f,g,a,b,c,d,e];
     rot([a,b,c,d,e,f,g],0) ↠ [a,b,c,d,e,f,g];
     rot([a,b,c,d,e,f,g],7) ↠ [a,b,c,d,e,f,g];
     rot([a,b,c,d,e,f,g],9) ↠ [c,d,e,f,g,a,b];
    rot([a,b,c,d,e,f,g],-3) ↠ [e,f,g,a,b,c,d];
   rot([a,b,c,d,e,f,g],-12) ↠ [c,d,e,f,g,a,b];
    rot([a,b,c,d,e,f,g],-7) ↠ [a,b,c,d,e,f,g];
    rot([a,b,c,d,e,f,g],-0) ↠ [a,b,c,d,e,f,g].
   ```

   Hint. It is helpful to put all the items around a circle, like markings on a clockwall, the first at noon, and imagine the only hand initially pointing to it. Then move the hand clockwise as many times as required by the rotation, or counter-clockwise if the offset is negative, and the final item pointed is the first item of the result. Then read the following items clockwise. Discuss delay and worst and best cases (if any). Provide a version in tail form. You will probably need the arithmetic operator `rem/2`, which gives the remainder of the Euclidian division, for instance,

`17 rem 3` $=$ `2`,   `-17 rem 3` $=$ `-2`,

`17 rem -3` $=$ `2`,   `-17 rem -3` $=$ `-2`.

Note how the sign of the remainder is always the sign of the dividend. This depends on the programming languages and must always be checked before doing arithmetics.

3. Transform into tail form and find the delay of the following definition. Transform it so the function returns a pair containing the number of rewrites to find the result.

```
flat_ter(L)           -> flat_ter(L,[]).
flat_ter(       [],B) -> rev(B);
flat_ter(   [[]|L],B) -> flat_ter(L,B);
flat_ter([[I|M]|L],B) ->
                             flat_ter(L,flat_ter([I|M],B));
flat_ter(    [I|L],B) -> flat_ter(L,[N|B]).
```

Have you seen the resulting definition before?

4. What is the delay of the following definition?

```
sflat_tf(L)       -> sflat(L,[],[]).
sflat(   [],[],B) -> rev(B);
sflat(   [], A,B) -> sflat(A,   [],    B);
sflat(  [I], A,B) -> sflat(I,    A,    B);
sflat([I|L], A,B) -> sflat(I,[L|A],    B);
sflat(    I, A,B) -> sflat(A,   [],[I|B]).
```

Chapter 7

Aliasing and Tail-Call Optimisation

We arbitrarily defined, on page 25, two distinct memory zones: the call stack and the heap. We would like now to provide a rationale for this scheme. Perhaps the best way is to forget now about the call stack and assume that everything, program and data, is stored in the heap, which is under the exclusive supervision of the garbage collector. The garbage collector can be thought of as an invisible manager which has full access to the contents of the memory at any moment and whose task consists in finding the nodes in the abstract syntax trees which are forever useless at some point during run-time. It consequently gets rid of them, so that subsequent nodes can find enough room in which to be stored. The remainder of this chapter will hopefully demonstrate that the concept of control stack, as a complement to the heap, arises naturally enough from a refined computational model solely based on the heap and that its purpose is a more efficient memory management.

We showed on page 72 how the technique used to find or estimate the delay of a call can also be applied to determine the number of pushes performed during the evaluation. This number gives, in general, an overestimation of how much memory needs to be allocated for the computation. Up to now, we assumed that the result of a rewrite step constitutes all the data in memory and, somehow, the function call is wholly replaced by the body of the first clause that matches it. In terms of abstract syntax trees, the tree on the left-hand side of the arrow is wholly replaced by the one on the right-hand side. For instance, let us recall the following definition

$\texttt{sum(\ \ [N])} \xrightarrow{\alpha} \texttt{N;}$
$\texttt{sum([N|L])} \xrightarrow{\beta} \texttt{N + sum(L).}$

and consider again the example in FIGURE 30 on page 160. Nothing is said about how nodes are created or erased. For instance, on the first rewrite, one node (`|`) has disappeared while a node (`+`) has been

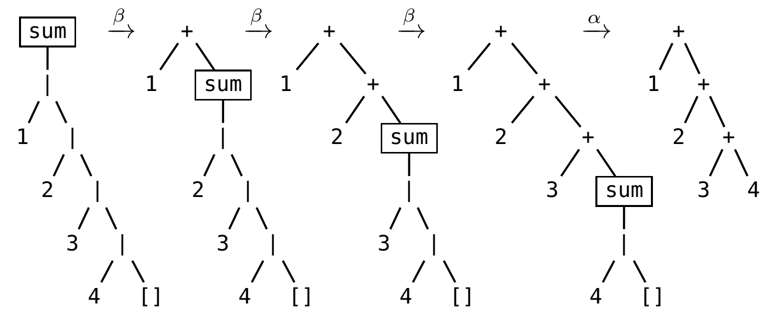


FIGURE 30: Computation of `sum([1,2,3,4])`

created and connected to a node `sum`. Of the latter, we don't know if it is the same as the one before the rewrite or whether a copy has been made and reconnected—because the argument is different. All these details are considered to be low-level, that is, they depend heavily on implementation issues, as how the run-time environment operates, and can be ignored if one only wants to understand the evaluation process and assess its delay. Nevertheless, when the concern about memory usage arises, for example, node creation and deletion, maximum amount of memory used etc., a more detailed view of the rewriting process is needed. This finer view relies on the abstract syntax trees, as they are a canonical representation of both data and programs. For example, the definition of `sum/1` is best interpreted in terms of rewrites on the abstract syntax trees in FIGURE 31 on page 160. It seems that the node (`|`) in the head of clause β can be freed, that is, reclaimed by the garbage collector, but the argument of `sum/1` may be in fact shared in the body where the function call is located by means of a variable with multiple occurrences. For example, let us imagine something like the following definition:

```
f(L) -> g(L,sum(L)).                        % Variable L is repeated.
```

If we assume that the value of the parameter `L` is not duplicated but, instead, that the two arguments `L` in the body refer to the same node

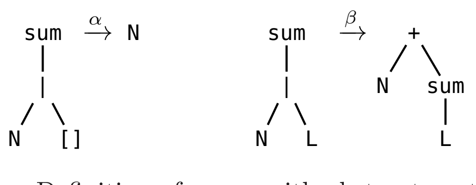


FIGURE 31: Definition of `sum/1` with abstract syntax trees

in memory, then the root of the abstract syntax tree of `L` cannot be deleted just after the rewrite step β because it is still necessary to compute the first argument of the call to `g/2` later. In general, it is best to assume that all values may be *shared* and, if not, the garbage collector will find out and delete the useless nodes. Consequently, when applying a rewrite rule, let us assume that the abstract syntax tree of the left-hand side remains in memory together with the right-hand side *and* that identical subtrees are shared by both, that is, are not duplicated. The garbage collector will detect and take care of useless nodes at unpredictable moments.

In other words, the rewrite rules do not explicitly delete any node and identical subtrees are shared. In order to visualise this data sharing, we need a kind of abstract syntax tree where identical subtrees occur only once, which supposes that some nodes can have two or more incoming edges. Of course, this cannot be a proper tree since, by definition, all nodes except the root must have exactly one parent. The adequate data structure which extends the abstract syntax tree with sharing is the finite *Directed Acyclic Graph* (DAG). Actually, a DAG is a collection of trees, called a *forest*, potentially sharing some of their subtrees. Therefore, such a graph has not, in general, a single root. In FIGURE 32 on page 161, we have the same definition of `sum/1` as in FIGURE 31 on the next page but with two DAGs, one for each clause, allowing maximum node sharing. Sharing is, as in FIGURE 29 on page 155 implemented by arrowed edges. Notice how this lower-level feature does not imply any explicit deletion of nodes or edges—which, at this level, can be thought of as references, or indirections, to nodes. Let us make use of this implementation of the definition of `sum/1` on the evaluation of the call `sum([3,7,5])`, whose first stages are shown in FIGURE 33 on page 162, where the full state of the memory is given as snapshots between dashed arrows and the last created node `sum` is framed. The arrows are dashed as to distinguish them from the ones in the definition, since clauses apply only to parts of the memory, in general, and we wanted to display all the data after each rewrite. Notice that all the nodes must be considered useful by the garbage collector.

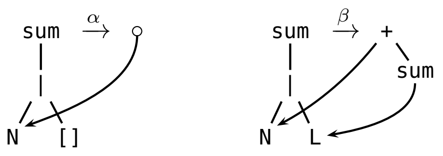


FIGURE 32: Definition of `sum/1` with maximum sharing

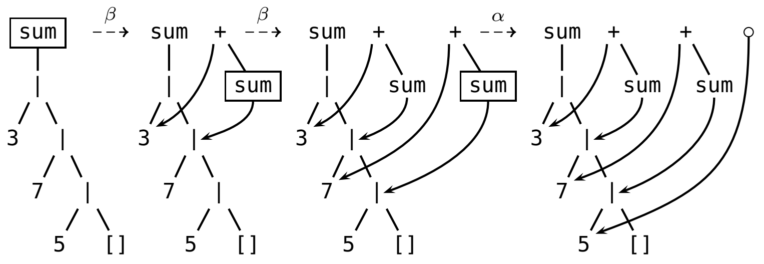


FIGURE 33: Computation of `sum([3,7,5])` with a DAG (phase 1/2)

This property is entailed if the *roots* of all the abstract syntax trees are deemed useful.

Naturally, this gives rise to the question on how the value of the original function call is finally computed. Examining FIGURE 33 on page 162 and comparing memory snapshots from left to right, we realise that roots (`+`) have been accumulating at the right of the original call, until a reference to a value has been reached—here, the integer `5`. This process is analogous to pushing items on a list, although a list containing not only values, but also expressions as `7 + sum([5])`. This invites us to perform the inverse operation, that is, popping items from the list in question, in order to finish the calculation. More precisely, we want to compute *values* from the trees composing the DAG from right to left, the roots of which are considered an item in a special list, until the tree of the original call is reached and associated with its own value—which is, by definition, the final result. A value can either be an *immediate* value like integers, atoms and empty lists, or a *constructed* value like non-empty lists and tuples. We may also deal with *references* to values, which are graphically represented by edges; for instance, the rightmost tree in the DAG is a reference to the immediate value `5`. *When the rightmost tree is an immediate value or a reference to a value, the second phase of the computation (leftward) can take place.* In the following, for the sake of conciseness, we shall write "value" when a "reference to a value" is also acceptable.

While our presently refined computational model forbids erasing nodes, because this is the exclusive task of the garbage collector, it does allow for the edge ending in the latest rewritten call to be overwritten by its value. As explained before, these calls have been successively framed in FIGURE 33 on the next page. The process therefore consists here in replacing the edge to the previous node `sum` with an edge to the current value. Then the patched tree is evaluated, perhaps leading to more trees to be pushed, and the process resumes.

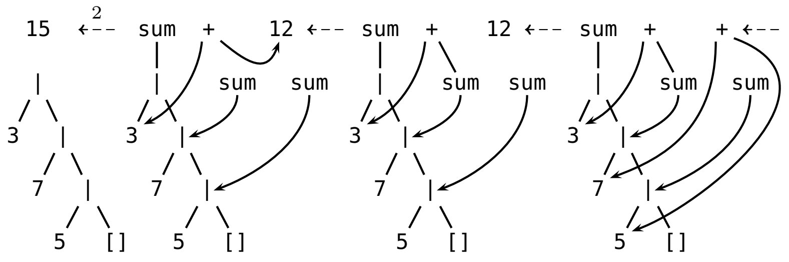


FIGURE 34: Computation of `sum([3,7,5])` (phase 2/2, heap-based)

This algorithm is shown at play in FIGURE 34 on page 163, which is to be read from right to left. The rightmost memory state is the result of the prior "pushing phase," on the last state of FIGURE 33. Note how all the nodes `sum` and (`+`) become useless, step by step. For illustration purposes, we made all the nodes (`+`) disappear as soon as possible and three nodes `sum` are reclaimed by the garbage collector, including the original one, that is, the leftmost. The leftmost dashed arrow has the superscript 2 because it combines two steps ($3+12 \rightarrow 15$ and discarding the original node `sum`) at once, for the sake of room. A node is useful if it can be reached from one of the roots of the DAG. Keep in mind that the argument of the original call, that is, `[3,7,5]`, may or may not be freed by the garbage collector, depending on it being shared or not (from outside the figure, that is, by some control context). The intermediary node containing the value `12` has been freed as well along the way. As far as the programmer is concerned, the moments when nodes are collected are arbitrary, as the garbage collector is exclusively in charge of this task, but we wanted to bring to the fore that garbage collection is interleaved with the computation or, if adhering to a multiprocessing view, we would say that collection and computation run in parallel, sharing the same memory space but without interferences from the programmer's point of view—only nodes that are forever unreachable from a point onwards during the execution are swept away.

The previous example is actually worth another, closer look. Indeed, we can predict exactly when the nodes `sum` can be reclaimed: after each step backwards (from right to left in FIGURE 34 on page 163), the rightmost node `sum` becomes useless. Same for the intermediary value `12`: it becomes unreachable from the roots of the DAG as soon is has been used to compute `15`. The same observation can be made about the nodes (`+`). All these facts mean that, in our example, we do not need to rely on the garbage collector to identify these nodes as useless: let us really implement a list of expressions, *separate from*

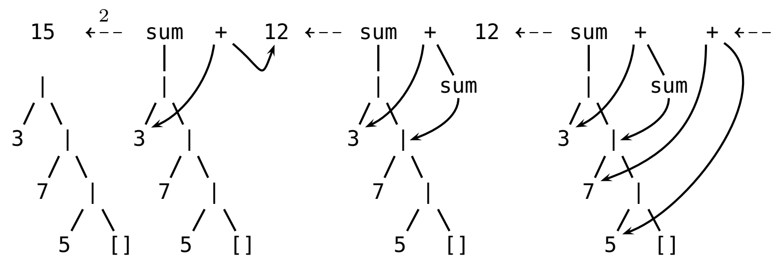


FIGURE 35: Computation of `sum([3,7,5])` (phase 2/2, stack and heap)

the heap, instead of solely relying on an analogy of push and pop and storing everything in the heap. This list is called the *control stack. For implementation reasons, the control stack never contains constructed data but references to constructed data.* As a consequence, a garbage collector is still needed for checking the constructed data—the heap and the control stack are complementary and make up the whole memory.

Consider how the computation in FIGURE 34 on page 163 can be improved with automatic deallocation of nodes based on a stack-based policy ("Last In, First Out") in FIGURE 35 on page 164. Remember that the value `[3,7,5]` is stored in the heap, not in the call stack, and that it may be shared. Also, due to space limitations on the page, the last step is actually twofold, as it was in FIGURE 34 on the facing page. We can seize the growth of the call stack in FIGURE 36 on page 164. The poppings, from right to left, are presented in FIGURE 37 on page 165. The corresponding algorithm consists in the following steps. Let us first assume, as a result of the previous steps, that the call stack is not empty and that the top item is a non-contructed value or a reference to a value, although we shall refer to them both as values.

1. While the call stack contains at least two objects, pop out the value, but without losing it, so another tree becomes the top;
   (a) if the root of the top tree is a node `sum`, then pop it and push instead the value;
   (b) else, the node `sum` in the tree has an incoming edge:

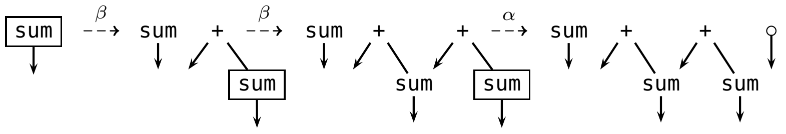


FIGURE 36: Call stack while computing `sum([3,7,5])` (phase 1/2)

   i. change its destination so it reaches the value and discard the node `sum`;
   ii. evaluate the patched top tree and resume the loop.
2. The only item remaining in the call stack is the result.

Actually, we allowed for atoms and integers to be stored in the call stack, so we could replace any tree which consists solely of a reference to such kind of value in the heap by a copy of the value. We can see in FIGURE 36 that the call stack grows at every step until a value is reached.

Let us investigate what happens when using an equivalent definition in tail form. We found on page 30:

$$
\begin{array}{lll}
\texttt{sum\_tf([N|L])} & \xrightarrow{\alpha} & \texttt{sum\_tf(L,N).}\\
\texttt{sum\_tf(\ \ \ [],A)} & \xrightarrow{\beta} & \texttt{A;}\\
\texttt{sum\_tf([N|L],A)} & \xrightarrow{\gamma} & \texttt{sum\_tf(L,A+N).}
\end{array}
$$

For the sake of clarity in FIGURE 38 on page 166, where we can follow the computation of `sum_tf([3,7,5])`, let us swap the arguments of `sum_-tf/2`, that is, we shall use the following equivalent definition instead:

$$
\begin{array}{lll}
\texttt{sum\_tf([N|L])} & \xrightarrow{\alpha} & \texttt{sum\_tf(N,L).}\\
\texttt{sum\_tf(A,\ \ \ [])} & \xrightarrow{\beta} & \texttt{A;}\\
\texttt{sum\_tf(A,[N|L])} & \xrightarrow{\gamma} & \texttt{sum\_tf(A+N,L).}
\end{array}
$$

FIGURE 38 on page 166 only shows the first phase, which consists in pushing in the call stack the tree newly produced by a clause and sharing the subtrees denoted by variables occurring both in the head and the body. The second phase consists in popping the accumulated roots in order to resume suspended control contexts and, in the end, only the final result remains in the control stack. In the case of `sum_tf/1`, we notice that we already found the result after the first phase: `15`. Therefore, in this case, the second phase does not contribute to build the value, which raises the question: why keep the previous trees in the first place? Indeed, they are useless and a common optimisation, named *tail-call optimisation* and implemented by compilers of functional languages, consists in popping the previous tree (matched by the head of the clause) and pushing the new one (created by the body of the clause). This way the control stack contains only one item at all times

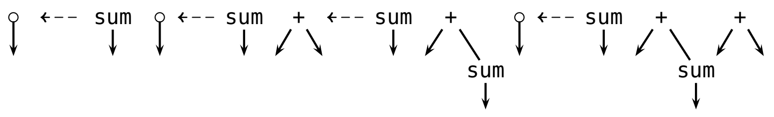


FIGURE 37: Call stack while computing `sum([3,7,5])` (phase 2/2)

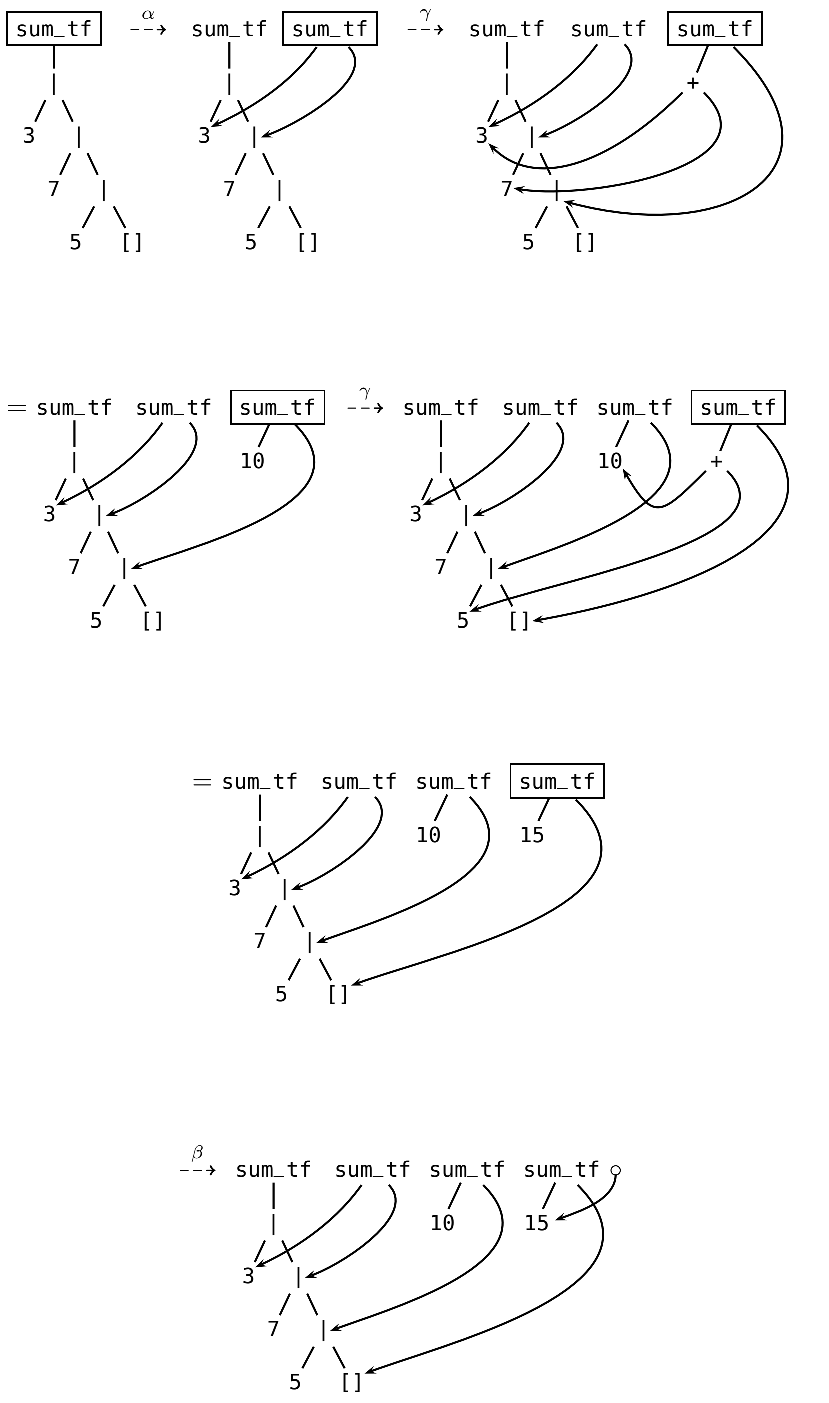


FIGURE 38: Computation of `sum_tf([3,7,5])`, no tail-call optimisation

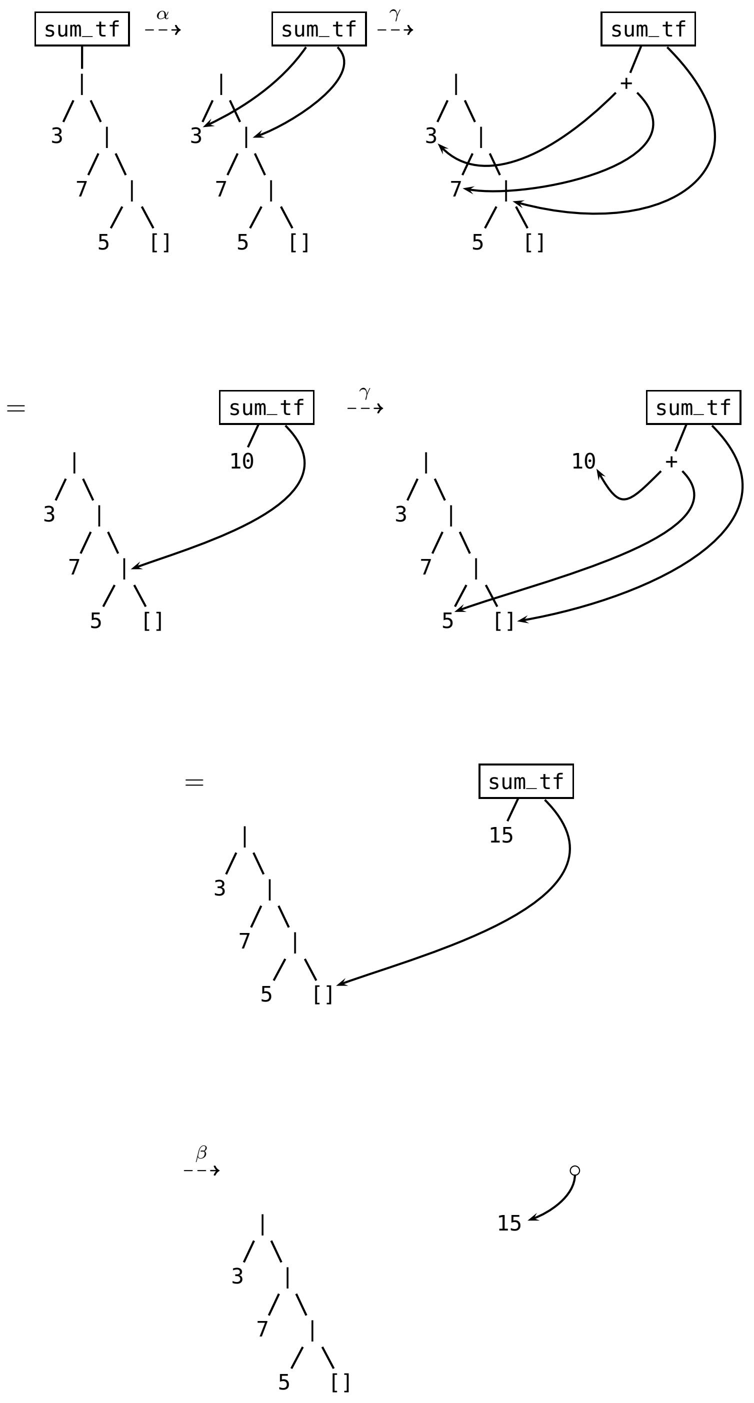


FIGURE 39: Computation of `sum_tf([3,7,5])`, tail-call optimisation

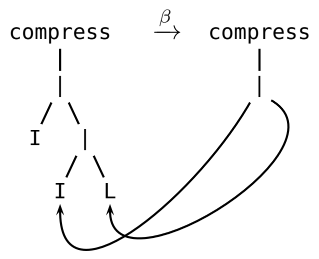


FIGURE 40: Directed acyclic graph (DAG) of clause β of `compress/1`

and, since we are using a stack, the garbage is collected immediately without the need for an external garbage collector, as in the heap. This optimisation is shown in FIGURE 39 on page 167 and should be contrasted with the series in FIGURE 38 on the following page. *Tail-call optimisation can be applied to all functions in tail form.*

Perhaps the previous discussions gave the impression that sharing is always maximum, that is, given a clause, the compiler will determine the largest abstract syntax subtrees common to the head and the body and create a directed acyclic graph (DAG) to implement this data sharing. That is not the case. Compilers, for example, the Erlang compilers, only rely on multiple occurrences of variables to define sharing, so it may be not maximum. Take for instance the function `compress/1` such that the call `compress(`L`)` results in a list identical to L, except that successively repeated items are reduced to one occurrence, for example, `compress([4,a,b,b,b,a,a])` is rewritten into `[4,a,b,a]`. One possible definition is the following:

```
compress(      []) -α-> [];
compress([I,I|L]]) -β-> compress([I|L]);
compress(   [I|L]) -γ-> [I|compress(L)].
```

In clause β, we can see that `[I|L]` both occurs in the pattern and the body because `[I,I|L]` is the same as `[I|[I|L]]`. In FIGURE 40 on page 168 is shown the abstract syntax view of clause β. Notice that the two occurrences of `I` in the left-hand side of the clause are not shared because we can only be certain of sharing when it is obtained by a known clause and the origin of the two `I`s is unknown. We can also see that no node (`|`) is shared, despite two lists sharing the same head `I` and same tail `L`. This shows that the sharing is not maximal and relies on the programmer. In order to increase the level of sharing, we need some special Erlang construct to express "This part of the pattern is given a name which can be used in the body." The notation we need is

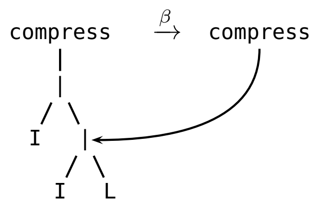


FIGURE 41: DAG of clause β of `compress/1` with maximum sharing

the operator (`=`) with a variable on its left-hand side and a pattern on its right-hand side. Consider the changes in bold:

```
compress(         []) -> [];
compress([I|L=[I|_]]) -> compress(L);          % L is an alias
compress(      [I|L]) -> [I|compress(L)].
```

The variable `L` is now an *alias* for the pattern `[I|_]`, so what the compiler understands now is displayed in FIGURE 41 on page 169. Note how the alias `L` does not occur in the figure, as it was used with the sole intent to have the edge representing the argument of `compress/1` connected to the second node (`|`), thus specifying that the entire sublist is shared and this without changing the meaning of the head of `compress/1`, that is, it matches the same input as before.

There is another definition, found on page 118, which would benefit from using an alias:

```
sflat(       []) -α-> [];
sflat(   [[]|L]) -β-> sflat(L);
sflat([[I|M]|L]) -γ-> join(sflat([I|M]),sflat(L));
sflat(    [I|L]) -δ-> [I|sflat(L)].
```

In clause γ, the list `[I|M]` is repeated in the head and the body, so the node (`|`) in it is not shared, thus incurring one useless node creation every time this clause is applied. Let us change it to

```
sflat(       [])   -α-> [];
sflat(   [[]|L])   -β-> sflat(L);
sflat([P=[_|_]|L]) -γ-> join(sflat(P),sflat(L));        % Alias P
sflat(      [I|L]) -δ-> [I|sflat(L)].
```

Now, the number of created nodes (`|`) is the number of non-list items in the input, or, in terms of the output, it is `len(sflat(`L`))`, where L is the input list to be flattened.

It is interesting to revisit previous definitions with a focus on memory use instead on the delay. Let us consider again the following definition of `join/2`:

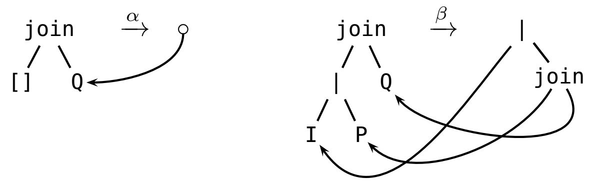


FIGURE 42: Definition of `join/2` with maximum sharing

```
join(   [],Q) →α Q;
join([I|P],Q) →β [I|join(P,Q)].
```

In FIGURE 42 on page 170, we can see the same definition making use of two DAGs in order to explicitly display node sharing. Let us detail the computation of `join([a,b],[c,d])` by evincing the node sharing and the call stack management. The first phase, consisting in pushing the newly created trees in the control stack is shown in FIGURE 43 on page 170.

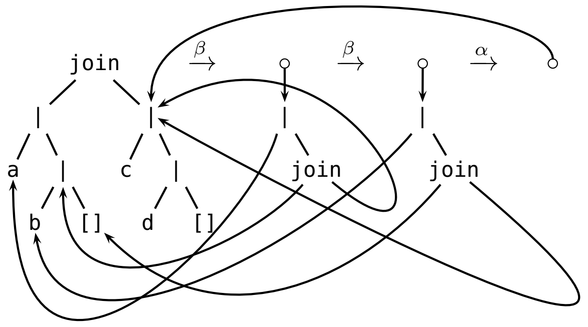


FIGURE 43: Computation of `join([a,b],[c,d])` (end of phase 1/2)

Note that, for the sake of room, we didn't use dashed arrows as before, but solid ones which are exactly the arrows used in the definition, thus the figure as a whole represents the last state of the control stack, its top being the rightmost value or reference to a value and its bottom being the leftmost call (the call originally evaluated). The second phase of the computation of `join([a,b],[c,d])` is shown in FIGURE 44 on page 171. It consists in replacing from right to left the reference to the previous call by the current (reference to a) value, until the initial call itself is removed and only the final result remains. Notice how the second argument, `[c,d]`, is actually shared with the output `[a,b,c,d]`.

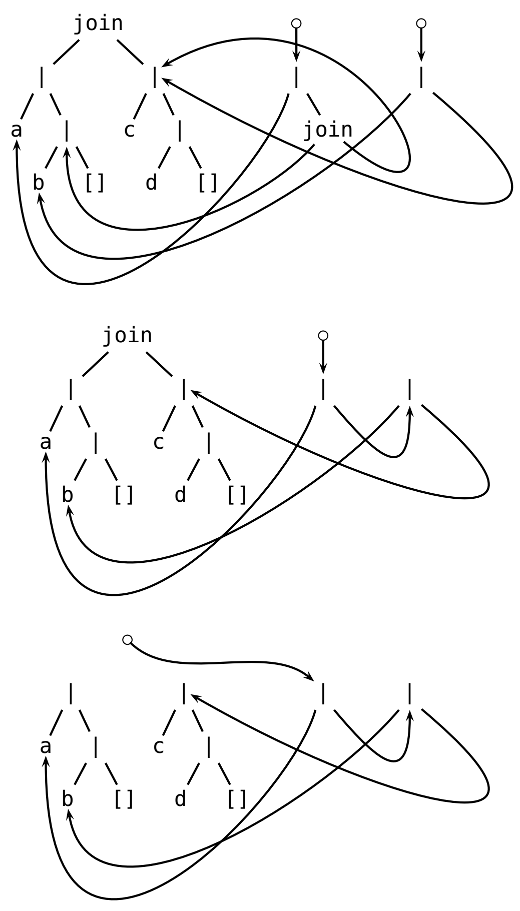


FIGURE 44: Computation of `join([a,b],[c,d])` (phase 2/2)

Let us go back to the definition of `rev/1`, as given on page 60:

```
rev(L)           -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

The last clause allocates a new node to hold `I`. In the end, whilst the list items are shared, the structure itself is not, that is, new nodes (`|`) are built. If the list contains n items, the delay is $n+2$ and the number of extra nodes is n. There is perhaps nothing surprising since these two lists are actually different, but if we reverse the reversed list, we end up with a list equal to the original *with a distinct, non-overlapping struc-*

ture. Therefore, reversing a list twice incurs a penalty both in terms of memory and time, although the input and the output are logically indistinguishable. This unfortunate situation may seem infrequent, but reversing twice *part of a list* has been done quite often up to this point. Think of a list accumulator in a definition in tail form.

Let us revisit `rm_fst/2`, as defined on page 80:

```
rm_fst(_,   []) -> [];
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].
```

This definition allows the output list to share the sub-list starting after the first occurrence of the item in the input list. This list is denoted by the variable `L` in the second clause. As a consequence, if the item occurs at position k, the head of the list being located at position 0, this function creates k extra nodes. Its worst case for the delay is the same as for memory consumption, that is, it happens when the item is absent from the list. Then the result is a list equal to the input, sharing its items with the input but not the structure, that is, the nodes (`|`).

What about `rm_fst_tf/2`, as found on page 85? We have

```
rm_fst_tf(I,L)    -> rm_fst(I,L,[]).
rm_fst(_,   [],A) -> rev(A);
rm_fst(I,[I|L],A) -> rev_join(A,L);
rm_fst(I,[J|L],A) -> rm_fst(I,L,[J|A]).
```

The accumulator `A` stores in reverse order the first items of the input which are not equal to the item that is searched. This list is finally reversed on top of the remaining part of the input, in order to build the result, so the situation is the same as with `rm_fst/2`, except that k more nodes have been created (two reversals are composed). This is too much if the item is missing in the list, because in this case the output is exactly the same as the input. What we want is to return the original list if the sequential search fails, instead of reversing the accumulator in the second clause. Hence, we need to add a parameter which is this original list and use it in case of failure:

```
rm_fst_tf(I,L)      -> rm_fst(I,L,[],L).
rm_fst(_,   [],_,P) -> P;                                   % Here
rm_fst(I,[I|L],A,_) -> rev_join(A,L);
rm_fst(I,[J|L],A,P) -> rm_fst(I,L,[J|A],P).
```

Incidentally, this change for the sake of memory improves also the delay in the case of a failure, since there is no more a reversal of a list whose size n is those of the initial list, thus saving $n + 2$ steps. Furthermore, this modification cannot be performed on `rm_fst/2`, on page 80:

```
rm_fst(_,   []) -> [];
rm_fst(I,[I|L]) -> L;
rm_fst(I,[J|L]) -> [J|rm_fst(I,L)].
```

If we try

```
rm_fst(P)         -> rm_fst(P,P).
rm_fst(_,   [],P) -> P;                                   % Error
rm_fst(I,[I|L],_) -> L;
rm_fst(I,[J|L],P) -> [J|rm_fst(I,L,P)].
```

we make a mistake because of the control context of the last clause (so the result is, wrongly, the original list *doubled*). Having a definition in tail form is a prerequisite for this kind of optimisation.

Let us also consider the original definition of `rm_lst/2`, as given on page 97:

```
rm_lst(I,L) -> rev(rm_fst(I,rev(L))).
```

At this point, we know that `rev(`L`)` always allocates n nodes if list L contains n items, and that `rm_fst(`I`,rev(`L`))` does as well in the worst case, which happens when I is missing in L. Under this assumption, `rev(rm_fst(`I`,rev(`L`)))` also allocates n nodes, because `rm_fst(`I`,rev(`L`))` holds n items. Therefore, `rm_lst(`I`,`L`)` creates $3n$ nodes in the worst case. It is worth noticing that the worst case for the delay coincides with the worst case for the memory consumption. The input and the output do not share any node (`|`), because of the reversals, only the items.

What about `rm_lst_tf(`I`,`L`)`, as found on page 105? We have

```
rm_lst_tf(I,L)        \xrightarrow{\alpha} rev_join(L,[],I).
rev_join(   [],Q,I)   \xrightarrow{\beta} rm_fst(I,Q,[]);
rev_join([J|P],Q,I)   \xrightarrow{\gamma} rev_join(P,[J|Q],I).
rm_fst(_,   [],A)     \xrightarrow{\delta} A;
rm_fst(I,[I|Q],A)     \xrightarrow{\epsilon} rev_join(Q,A);
rm_fst(I,[J|Q],A)     \xrightarrow{\zeta} rm_fst(I,Q,[J|A]).
```

Clause α does not allocate in the heap. Clause γ creates one node (`|`), that is, n nodes if the input list contains n items. Clause β performs no allocation. We found that the worst case for the delay occurs when the item `I` is present *anywhere* in the input list. How does that translates to pushes? Remarkably, one rewrite by means of clause ζ results in exactly one node allocation. Same for the recursive clause of `rev_join/2` (see definition above). Therefore, the worst case in terms of delay is also the worst case in terms of memory allocation. Let us assume arbitrarily, but without loss of generality, that the item is the head of the input list. Then clause ζ is unused and `rev_join/2` performs n node creations in the body of clause ϵ. So the total number of node allocations is $2n$ in the worst case, of which n hold the result and n are temporary nodes which can be reclaimed by the garbage collector.

Consider another example. We want to make a list containing the head of a given list repeated as many times as there are items in the given list. For instance, `[b,3,1]` results in `[b,b,b]`. There are at least two ways to implement this effect, with the same

```
rep_fst1(   []) -> [];
rep_fst1([I|L]) -> [I|repeat(I,L)].
repeat(_,   []) -> [];
repeat(I,[_|L]) -> [I|repeat(I,L)].
```

and

```
rep_fst2(     []) -> [];
rep_fst2(    [I]) -> [I];
rep_fst2([I,_|L]) -> [I|rep_fst2([I|L])].
```

Both functions have delay $n + 1$. The difference lies in the memory used: the last clause of `rep_fst2/1` performs two pushes instead of one for the last clause of `repeat/2`. As a consequence, `rep_fst2/1` allocates $2(n-1)+1 = 2n-1$ nodes if $n \neq 0$, of which $n-1$ are temporary. By contrast, `rep_fst1/1` only allocates n nodes.

Exercise. [See answer page 378.]

Consider now the definition of `srev/1`:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

Determine what part of the input is shared with the output. Is there a best and worst case? If so, are they different from the delay cases?

Chapter 8

Persistence and Backtracking

A distinctive feature of purely functional data structures is that they are *persistent*. Persistence is the property of values to remain constant over time. It is a consequence of Erlang functions always creating a new version of a data structure to be updated, instead of modifying it in place. Moreover, as we learnt previously, only the nodes which actually differ between the input and the output are actually created, whereas the others are shared by means of aliases or variables occurring both in the pattern and the body of a clause. Sharing is sound because it relies on persistence: since the data cannot be modified in place from different access points in the program (the roots of the directed acyclic graph), there is no way to tell apart a perfect copy from a reference to the original—in a chronological sense. In this chapter, we show how persistence allows us to undo operations, that is, *to backtrack*, quite straightforwardly and how sharing makes it affordable in terms of memory consumption. As a simple case study, let us consider a list whose different versions we want to record, as items are pushed in and popped out, starting from the empty list.

Version-based persistence. A simple idea to implement data structures enabling backtracking consists in keeping all successive states; then, backtracking means accessing a previous version. A list can be used to keep such a record, called *history*. Since, from an algorithmic standpoint, a list is a stack, the most recent version is the top and is thus accessed in constant time. The older a version is, the longer it takes to access it, but always in proportion to the time elapsed in-between. Furthermore, two successive versions share as much structure as the clauses that generated the new one specify. As a consequence, it is affordable in terms of memory to thread the history, instead of only passing around the latest version alone. Consider the following definitions of `push/2` and `pop/1`. The call `push(`I`,`H`)` evaluates in a history

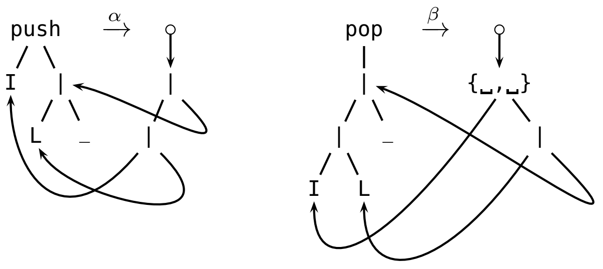


FIGURE 45: Definition of `push/2` and `pop/1` with DAGs

which is made of history H on top of which the last version, extended with I, has been pushed. The call `pop(`H`)` is rewritten in a pair whose first component is the item I on the top of the last version in H and second component is H on top of which the last version without I is pushed. In discussing recursive definitions, we may also refer to the *current version*, meaning the version corresponding to the input of the clause at hand. Let us assume in the following examples that the history always contains an empty list at the beginning, so it is never empty. We write

```
push(I,H=[L|_])  -α-> [[I|L]|H].
pop(H=[[I|L]|_]) -β-> {I,[L|H]}.
```

Note the crucial use of *aliases* (`H`) to guarantee that the history before the change is not duplicated by the function calls. Imagine now the following series of operations on a list originally empty: push `a`, push `b`, push `c`, pop, push `d`. Let us ignore the popped item by means of a function projecting the second component of a pair:

```
snd(_,Y) -> Y.
```

Then the whole history of the operations can be obtained by the following composition:

```
push(d,snd(pop(push(c,push(b,push(a,[])))))).
```

This expression evaluates, as expected, into

```
[[d,b,a],[b,a],[c,b,a],[b,a],[a],[]].
```

In order to understand how the data and the structure is shared among versions in this example, let us reconsider the clauses above under the aspect of the Directed Acyclic Graphs (DAG) built upon the abstract syntax trees in FIGURE 45 on page 176. The graph of the result is shown in FIGURE 46 on page 177. The kth previous version in history H is `ver(`k`,`H`)`:

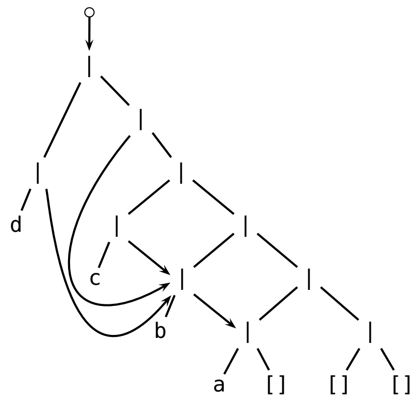


FIGURE 46: DAG of `[[d,b,a],[b,a],[c,b,a],[b,a],[a],[]]`

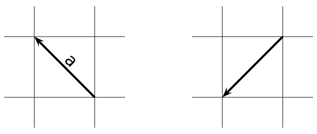
```
ver(K,H) when K >= 0 -> ver__(K,H).

ver__(0,[V|_]) -> V;
ver__(K,[_|H]) -> ver__(K-1,H).
```

It seems interesting to remark that the technique presented allows the newest version in the history to be modified, but not the older ones: just pop the head of the history and push some other version:

```
change_last(Update,[Last|History]) -> [Update|History].
```

When a data structure allows every version in its history to be modified, it is said *fully persistent*; if only the newest version is updatable, it is said *partially persistent*.

Update-based persistence. In order to achieve full persistence, we could keep on record the list of changes to the data structure, instead of the successive states, as we just did. In our continued example, we have two kinds of updates: `{push,`I`}`, where I is some item to push, and `pop`. Now history is a list of such updates. But not all series of `push` and `pop` are valid, a trivial example being `[pop]`, which does not allow a version to be extracted. Another is `[pop,pop,{push,a}]`. How can we caracterise the valid histories? It is useful to consider a graphical representation of `push` and `pop`, as proposed in FIGURE 47 on page 178. A history is, graphically, a broken line composed with these two kinds of segments, oriented from right to left in order to follow the Erlang syntax for lists. Consider for instance the representation of the history `[pop,pop,{push,d},{push,c},pop,{push,b},{push,a}]` in FIGURE 48 on page 178. Horizontally are the updates and vertically the lengths of the



(a) {push,a} (b) pop

FIGURE 47: Graphical representations of updates

versions. The line starts at the origin of the axes because the version at that point is supposed to be an empty list (the caller may change this assumption at her own expenses). It ends at the present moment, distinguished by a dot, where the latest version has length 1. *A valid history is a line which never crosses the absissa axis.* The definition of `push/2` is straightforward:

```
push(I,H) -> [{push,I}|H].
```

The design of `pop/1` requires more care as

```
pop(H) -> [pop|H].                                   % Incomplete
```

does not return the item supposed to be popped and also may create an impossible configuration of the data structure, as `pop([])` is accepted instead of being rejected. A better try is

```
pop(H=[{push,I}|_]) -> {I,[pop|H]}.            % Still incomplete
```

but this definition is incomplete, as it doesn't handle valid histories like

`pop([pop,{push,b},{push,a}])` $\not\twoheadrightarrow$.

The missing case is thus when the last update is a `pop`. It is not neces-

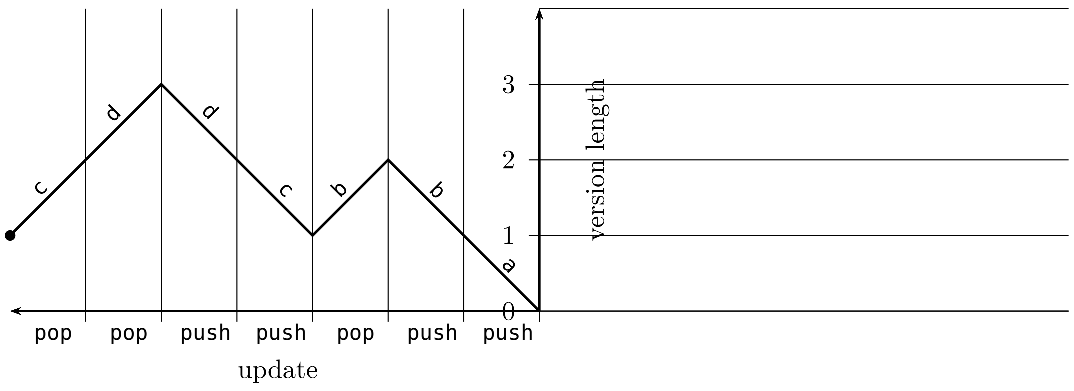


FIGURE 48: `[pop,pop,{push,d},{push,c},pop,{push,b},{push,a}]`

sary to extract the newest version in order to take its topmost item: it is sufficient to deduce this item from the updates. If it exists, we know that we can record a `pop` update because the resulting version is constructible. Graphically, it means that the historical line reached a point of ordinate 1 at least, so a `pop` will not cross the absissa axis. The task of finding this item is delegated to the function `top/1`, hence

```
pop(H) -> {top(H),[pop|H]}.                              % Complete
```

Note that the alternative `[{pop}|H]` would waste memory for one node. The algorithm implemented by `top/1` is best understood on the graphical representation of updates. Consider again our running example in FIGURE 48 on page 178. The last update was a `pop`. What is the topmost item of the latest version? By following the historical line backwards, that is, from left to right, we see that the penultimate update was also a `pop`, so we cannot conclude yet. The antepenultimate update is a `push`, but its associated item, `d`, cannot be the item we are looking for because it is popped by the following update (which is the penultimate). Following back in time we find another `push`, but its item, `c`, is popped by the last update. The previous update is a `pop`, so we are back to square one. Following backwards is a `push` but its item, `b`, is popped by the next update. Finally, the previous update is a `push` and its item, `a`, is the one we want, because it is the first we found which is not popped by further updates. Graphically, this update is easily found by drawing a horizontal line from the present version to the right. There are several intersections with the historical line. The one we are interested in is the first end of a `push` update. If such a point does not exist, we reach the origin. It means that the last version (the leftmost point of the line) is an empty list and, consequently, the `pop` update must fail, as expected. Consider in FIGURE 49 on page 180 the geometrical solution. This results in the following code:

```
top(H)               -> top(0,H).
top(0,[{push,I}|_]) -> I;
top(K,[{push,_}|H]) -> top(K-1,H);
top(K,     [pop|H]) -> top(K+1,H).
```

The counter `K` is used to keep track of the number of `pop` updates (or, equivalently, the height of the old versions with respect to the last one), so, when it reaches `0`, we check whether the current update is a `push` (we know that we are on the same horizontal line as the last version). If so, we are done; otherwise, we resume the search. Encountering the origin leads to a match failure. Now, the call

```
push(d,snd(pop(push(c,push(b,push(a,[])))))))
```

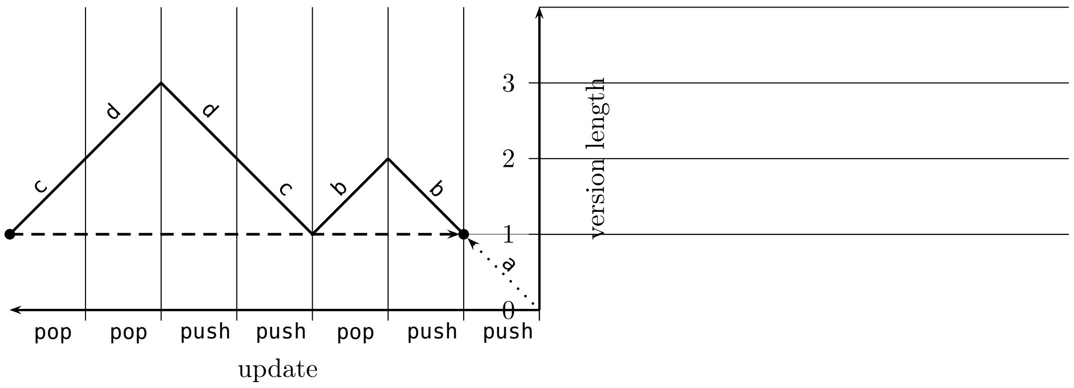


FIGURE 49: Finding the topmost item a of the last version

evaluates, as expected, into

```
[{push,d},pop,{push,c},{push,b},{push,a}].
```

Let us pop twice more:

```
pop(snd(pop(push(d,snd(pop(push(c,push(b,push(a,[])))))))))
```

and the evaluation yields the expected pair

```
{b,[pop,pop,{push,d},pop,{push,c},{push,b},{push,a}]}.
```

Let `ver`(k,H) be the kth version from the last moment in history H, counting the last version as 0. When a past version is needed, that is, we walk back through history until the time requested (searching phase),

```
% Searching
ver(0,    H) -> last(H);
ver(K,[_|H]) -> ver(K-1,H).
```

and we rebuild the version from that point (building phase). How do we define `last/1`, which builds the last version? There are several ways. Perhaps the first idea is to recursively construct the penultimate version and if the last update was a push, then perform a push on it, otherwise, its top item is popped:

```
% Building
last(          []) -> [];
last([{push,I}|H]) -> [I|last(H)];
last(     [pop|H]) -> tail(last(H)).
tail([_|L])       -> L.
```

Note how `tail/1` is used to discard the popped item on our way, but this creates an asymmetry in the delay between the processing of a push update and a pop (one more rewrite). Let us determine the delays of `ver/2` and `last/1`. First, let us note $\mathcal{D}^{\mathsf{ver}}_{k,n}$ and $\mathcal{D}^{\mathsf{last}}_{p}$ the respective delays of `ver(`k`,`H_1`)` and `last(`H_2`)`, where histories H_1 and H_2 have respective lengths n and p. The Erlang definition of `ver/2` directly leads to $\mathcal{D}^{\mathsf{ver}}_{k,n} = (k+1) + \mathcal{D}^{\mathsf{last}}_{n-k}$. The delay of `last/1` depends on the nature of the updates: if a `push`, the second clause adds a delay of 1, otherwise the third clause adds a delay of 2, due to the additional delay of calling `tail/1`. Formally:

$$\mathcal{D}^{\mathsf{last}}_{0} = 1, \qquad \mathcal{D}^{\mathsf{last}}_{p+1} = \begin{cases} 1 + \mathcal{D}^{\mathsf{last}}_{p} & \text{if } \mathtt{push}, \\ 2 + \mathcal{D}^{\mathsf{last}}_{p} & \text{if } \mathtt{pop}. \end{cases}$$

This asymmetry can be remedied by a special construct of Erlang, commonly found in other programming languages: a *case*, also known as a *switch* in Java and C:

```
last(          []) -> [];
last([{push,I}|H]) -> [I|last(H)];
last(     [pop|H]) -> case last(H) of [_|V] -> V end.
```

The meaning is "Compute the value of `last(H)`, which must be a non-empty list. Let us name `V` its tail and `V` is the value of the body." The arrow in the case construct is not compiled as a function call and thus its delay is 0. When exactly one case of the value being tested (here `last(H)`) is expected, like here, Erlang permits a shortcut notation:

```
last(          []) -> [];
last([{push,I}|H]) -> [I|last(H)];
last(     [pop|H]) -> [_|V] = last(H), V.
```

Now it is very easy to derive the delay of `last(H)`: $\mathcal{D}^{\mathsf{last}}_{p} = p + 1$. Hence $\mathcal{D}^{\mathsf{ver}}_{k,n} = (k+1) + \mathcal{D}^{\mathsf{last}}_{n-k} = k + 1 + (n-k) + 1 = n + 2$. Interestingly, the delay does not depend on what version is computed. What about memory usage? As usual, since we do not know anything about the strategy of the garbage collector, we cannot deduce the exact amount of memory necessary to compute the result, but we can count the number of pushes needed, which allows us to compare different versions of the same function. (Incidentally, this number can be considered a contribution to the overall delay, as creating a node takes some time, therefore it complements our delay analysis as it only accounts for the number of function calls.) The only clause of `last/1` performing a push is the second one and it applies when a `push` update is found. Therefore, the number of push nodes created is the number of `push` updates in the history. This is a waste in certain cases, for example, when the version built is empty (consider the history `[pop,{push,6}]`). The optimal situation would be to allocate only as much as the computed version actually contains. This brings us back to the function `top/1`, which computes the top of the last version. What we need is to call it recursively on the remainder of the history, until the empty list is reached. Here is the modification:

```
ver(0,    H) -> last(0,H);
ver(K,[_|H]) -> ver(K-1,H).

last(0,[{push,I}|H]) -> [I|last(0,H)];         % Keep going back
last(M,[{push,_}|H]) -> last(M-1,H);
last(M,      [pop|H]) -> last(M+1,H);
last(_,           []) -> [].                   % Origin of history
```

We still have $\mathcal{D}_p^{\mathsf{last}} = p + 1$, but the number of push nodes created is now the length of the last version `ver(0,`H`)`. This is an example of *output-dependent* measure. There is still room for some improvement in the special case when the historical line meets the absissa axis—in other words, when a past version is the empty list. There is no need to visit the updates *before* a `pop` resulting into an empty version, for instance, in figure 50 on page 182, it is useless to go past `{push,c}` to find the last version to be `[c]`. In order to detect whether the historical line meets the absissa axis, let us keep, jointly with the history of updates, the length of the last version. First, let us modify `push/2` and `pop/1` accordingly:

```
push(I,{N,H}) -> {N+1,[{push,I}|H]}.
pop({N,H}) -> {top(H),{N-1,[pop|H]}}.
```

We have to rewrite `ver/2` to keep track of the length of the last version

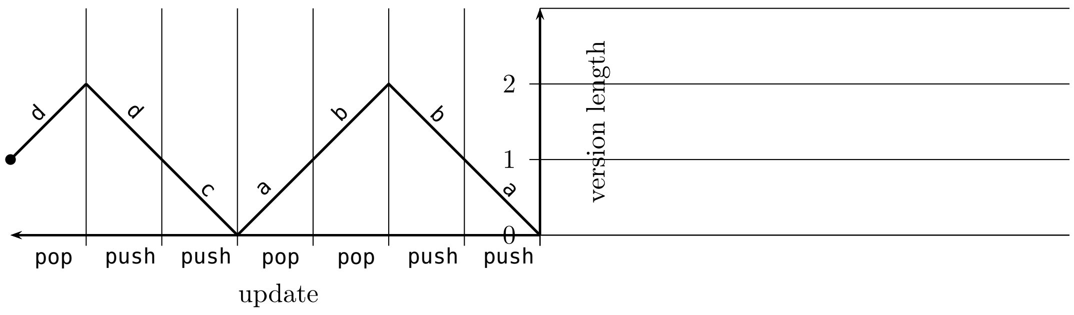


figure 50: Last version `[c]` found in four steps

and, while we are at it, we can check that `K` is not negative:

```
ver(K,NH) when K >= 0     -> ver__(K,NH).
ver__(0,              NH) -> last(0,NH);
ver__(K,{N,     [pop|H]}) -> ver__(K-1,{N+1,H});
ver__(K,{N,[{push,_}|H]}) -> ver__(K-1,{N-1,H}).
```

We can reduce the memory footprint by separating the length of the current version `N` from the current history `H`, so no pair nodes (`{␣,␣}`) are allocated:

```
ver(K,{N,H}) when K >= 0 -> ver__(K,N,H).
ver__(0,N,           H)  -> last(0,N,H);
ver__(K,N,     [pop|H])  -> ver__(K-1,N+1,H);
ver__(K,N,[{push,_}|H])  -> ver__(K-1,N-1,H).
```

Let us change the definition of `last/2` so it processes the length of the current version:

```
last(_,0,           _) -> [];                      % Absissa 0
last(0,N,[{push,I}|H]) -> [I|last(0,N-1,H)];
last(M,N,[{push,_}|H]) -> last(M-1,N-1,H);
last(M,N,     [pop|H]) -> last(M+1,N+1,H).
```

Notice how the last clause of the previous version of `last/2`

```
last(_,          []) -> [].                      % Useless now
```

became redundant with the new first clause and therefore removed. The delay now presents a worst and best cases. The worst case is when the bottom item of the last version is the first item pushed in the history, so `last/2` has to recur until the origin of time. This incurs the same delay as previously: $\mathcal{W}_p^{\mathsf{last}} = p + 1$. The best case happens when the last version is empty. In this case $\mathcal{B}_p^{\mathsf{last}} = 1$, and this is an occurrence of the kind of improvement we sought.

Changing the past. The *update-based approach* is fully persistent, that is, it allows us to modify the past as follows: traverse history until the required moment, pop the update at that point, push another one and simply put back the previously traversed updates, which must have been kept in some accumulator. It is of utmost importance that, by changing the past, we do not create a history with a non-constructible version in it, that is, we must check that the historical line does not cross the absissa axis after the modification. If the change consists in replacing a `pop` by a `push`, there is no need to worry, as this will raise by 2 the end point of the line. It is the converse change that requires special attention, as this will lower by 2 the end point. The ± 2 offset comes from the vertical difference between the ending points of

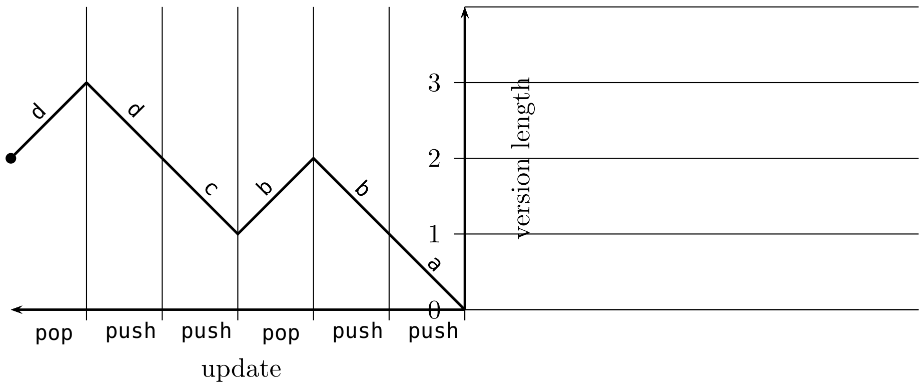


FIGURE 51: `[pop,{push,d},{push,c},pop,{push,b},{push,a}]`

a `push` and a `pop` update of same origin, as can be easily figured out by looking back at FIGURE 47 on the following page. As a consequence, in FIGURE 48 on the next page, the last version has length 1, which implies that it is impossible to replace a `push` by a `pop`, anywhere in the past. Let us consider the history in FIGURE 51 on page 184 Let us note `change(`k`,`U`,{`n`,`H`})`, where k is the index of the update we want to change, indexing the last one at 0; U is the new update we want to set; n is the length of the last version of history H. The call

```
change(3,{push,e},
       {2,[pop,{push,d},{push,c},pop,{push,b},{push,a}]})
```

is rewritten into (changes in bold)

```
{4,[pop,{push,d},{push,c},{push,e},{push,b},{push,a}]})
```

This call succeeds because, as we can see graphically in FIGURE 52 on page 185, the new historical line does not cross the absissa axis. We can see in FIGURE 53 on page 185 the result of the call

```
change(2,pop,
       {2,[pop,{push,d},{push,c},pop,{push,b},{push,a}]}).
```

It should be clear now that

```
change(4,pop,
      {2,[pop,{push,d},{push,c},pop,{push,b},{push,a}]})↛,
change(5,pop,
      {2,[pop,{push,d},{push,c},pop,{push,b},{push,a}]})↛
```

because the line would cross the absissa axis. All these examples help in guessing the characteristic property for a replacement to be valid:

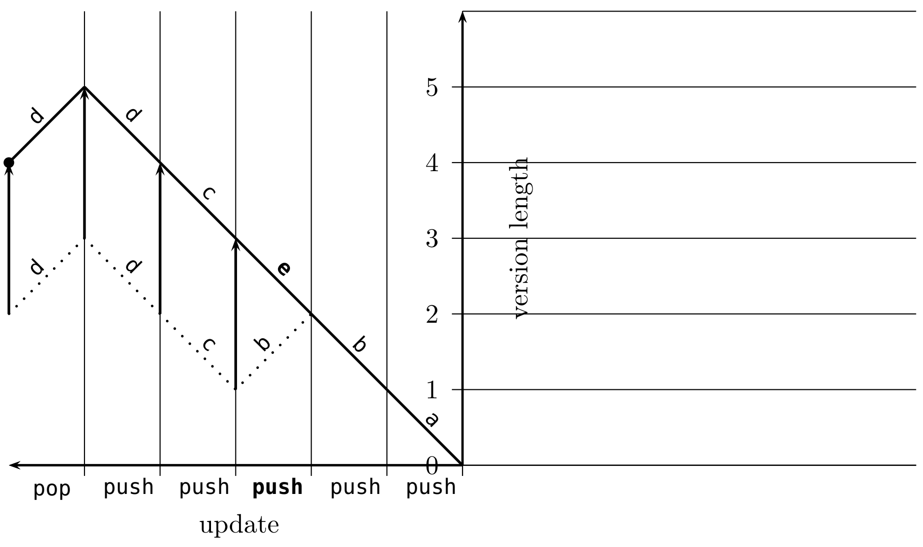


FIGURE 52: Changing `pop` (`b`) into `{push,e}`

- the replacements of a `pop` by a `push`, a `pop` by a `pop`, a `push` by a `push` are always valid;
- the replacement of a `push` by a `pop` at update $k > 0$ is valid if and only if the historical line between updates 0 and $k-1$ remains above or reaches without crossing the horizontal line of ordinate 2.

We can divide the algorithm in two phases, as we did with `ver/2` and `ver__/3`, on the one hand, and `last/3`, on the other hand. First, the

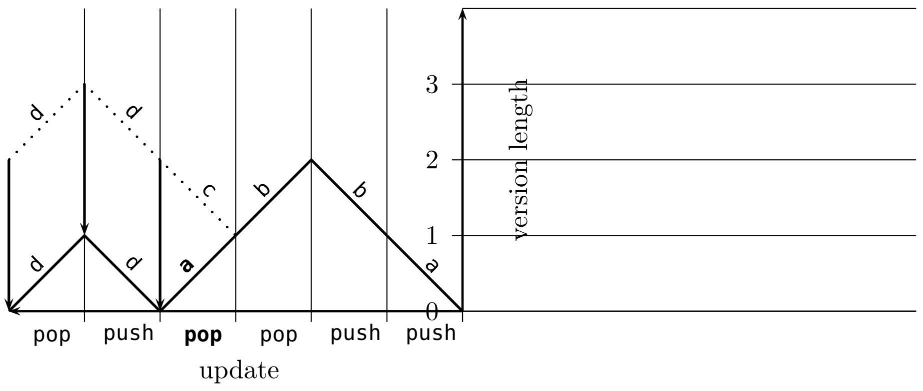


FIGURE 53: Changing `{push,c}` into `pop` (`a`).

update to be replaced must be found, but, the difference with `ver/2` (or, more accurately, `ver__/3`), is that we may need to know if the historical line before reaching the update lies above the horizontal line of ordinate 2. This is easy to check if we maintain across recursive calls the lowest ordinate reached by the line. The second phase performs the change of update and checks if the resulting history is valid. Let us implement the first phase. First, we check that the index of the update is not negative; the length of the last version is separated from the history, in order to save some memory space, and the lowest ordinate is this length, which we pass as an additional argument to another function, `chg/5`:

```
change(K,U,{N,H}) when K >= 0 -> chg(K,U,N,H,N).
```

Function `chg/5` traverses `H` while decrementing `K`, until the latter reaches `0`. At the same time, the length of the current version is computed (third argument) and compared to the previous lowest ordinate (the fifth argument), which is updated according to the outcome. When the update to be changed is found, `K` is `0`.

```
chg(0,U,N,          H,M)          -> ▭;
chg(K,U,N,     [pop|H],M)         -> chg(K-1,U,N+1,H,M);
chg(K,U,N,[{push,_}|H],M) when M<N -> chg(K-1,U,N-1,H,M);
chg(K,U,N,[{push,_}|H],_)         -> chg(K-1,U,N-1,H,N-1).
```

The problem is that we forget the history down to the update we are looking for. There are two methods to record it: either we use an accumulator and stick with a definition in tail-form, or we put back a visited update after each return of recursive call. The latter is faster, as there is no need to reverse the accumulator when we are done; the former allows us to share the history up to the update, at the cost of an extra parameter being the original history. Let us opt for limiting the number of arguments:

```
chg(0,U,_,             H,M)           -> repl(U,H,M);
chg(K,U,N,       [pop|H],M)           ->
            {N1,H1} = chg(K-1,U,N+1,H,  M), {N1,[pop|H1]};
chg(K,U,N,[P={push,_}|H],M) when M < N ->
            {N1,H1} = chg(K-1,U,N-1,H,  M), {N1,  [P|H1]};
chg(K,U,N,[P={push,_}|H],_)           ->
            {N1,H1} = chg(K-1,U,N-1,H,N-1), {N1,  [P|H1]}.
```

Notice how the length `N1` of the changed history is invariant, because we can simply make it out once the update to change is found:

- replacing a `pop` by a `pop` or a `push` by a `push` leaves the original length invariant;

- replacing a pop by a push increases the original length by 2;
- replacing a push by a pop, assuming this is valid, decreases the original length by 2.

This task is up to the new function repl/3 ("replace"), which implements the second phase (the replacement itself). The idea is that it returns a pair made of the differential in length D and the new history H1, which implies that we need to change change/3:

```
change(K,U,{N,H}) when K >= 0 ->
                                {D,H1} = chg(K,U,N,H,N), {N+D,H1}.

repl(       pop,   H=[pop|_],_)            -> { 0,      H};
repl(P={push,_},[{push,_}|H],_)            -> { 0,  [P|H]};
repl(P={push,_},     [pop|H],_)            -> { 2,  [P|H]};
repl(       pop,[{push,_}|H],M) when M > 1 -> {-2,[pop|H]}.
```

Improvement. Just like on page 151 we saved memory by nesting tuples instead of putting tuples in a list when transforming systematically definitions into tail form, we can here save memory by implementing a history by nested updates. Instead of having

```
[pop,pop,{push,d},pop,{push,c},{push,b},{push,a}]
```

we shall prefer

```
{pop,{pop,{push,d,{pop,{push,c,{push,b,{push,a,{}}}}}}}}
```

This alternative encoding leads to save a node and an edge for each push update. We wrote

```
-module(pers).
-export([push/2,pop/1,ver/2,change/3]).

push(I,{N,H}) -> {N+1,[{push,I}|H]}.
pop({N,H}) -> {top(H),{N-1,[pop|H]}}.

top(H)               -> top(0,H).
top(0,[{push,I}|_]) -> I;
top(K,[{push,_}|H]) -> top(K-1,H);
top(K,     [pop|H]) -> top(K+1,H).

ver(K,{N,H}) when K >= 0 -> ver__(K,N,H).
ver__(0,N,           H)  -> last(0,N,H);
ver__(K,N,     [pop|H])  -> ver__(K-1,N+1,H);
ver__(K,N,[{push,_}|H])  -> ver__(K-1,N-1,H).

last(_,0,           _) -> [];
last(0,N,[{push,I}|H]) -> [I|last(0,N-1,H)];
last(M,N,[{push,_}|H]) -> last(M-1,N-1,H);
last(M,N,     [pop|H]) -> last(M+1,N+1,H).

change(K,U,{N,H}) when K >= 0 ->
                         {D,H1} = chg(K,U,N,H,N), {N+D,H1}.

chg(0,U,_,             H,M)            -> repl(U,H,M);
chg(K,U,N,       [pop|H],M)            ->
             {N1,H1} = chg(K-1,U,N+1,H,  M), {N1,[pop|H1]};
chg(K,U,N,[P={push,_}|H],M) when M < N ->
             {N1,H1} = chg(K-1,U,N-1,H,  M), {N1,  [P|H1]};
chg(K,U,N,[P={push,_}|H],_)            ->
             {N1,H1} = chg(K-1,U,N-1,H,N-1), {N1,  [P|H1]}.

repl(       pop,   H=[pop|_],_)            -> { 0,      H};
repl(P={push,_},[{push,_}|H],_)            -> { 0,  [P|H]};
repl(P={push,_},     [pop|H],_)            -> { 2,  [P|H]};
repl(       pop,[{push,_}|H],M) when M > 1 -> {-2,[pop|H]}.
```

Now we can proceed to improve the definitions as intended:

```
-module(pers_opt).
-export([push/2,pop/1,ver/2,change/3]).

push(I,{N,H}) -> {N+1,{push,I,H}}.
pop({N,H}) -> {top(H),{N-1,{pop,H}}}.

top(H)            -> top(0,H).
top(0,{push,I,_}) -> I;
top(K,{push,_,H}) -> top(K-1,H);
top(K,   {pop,H}) -> top(K+1,H).

ver(K,{N,H}) when K >= 0 -> ver__(K,N,H).
ver__(0,N,           H)  -> last(0,N,H);
ver__(K,N,     {pop,H})  -> ver__(K-1,N+1,H);
ver__(K,N,  {push,_,H})  -> ver__(K-1,N-1,H).

last(_,0,         _) -> {};
last(0,N,   {pop,H}) -> {pop,last(0,N-1,H)};
last(0,N,{push,I,H}) -> {push,I,last(0,N-1,H)};
last(M,N,{push,_,H}) -> last(M-1,N-1,H);
last(M,N,   {pop,H}) -> last(M+1,N+1,H).

change(K,U,{N,H}) when K >= 0 ->
                        {D,H1} = chg(K,U,N,H,N), {N+D,H1}.

chg(0,U,_,         H,M)            -> repl(U,H,M);
chg(K,U,N,   {pop,H},M)            ->
          {N1,H1} = chg(K-1,U,N+1,H,  M), {N1,   {pop,H1}};
chg(K,U,N,{push,I,H},M) when M < N ->
          {N1,H1} = chg(K-1,U,N-1,H,  M), {N1,{push,I,H1}};
chg(K,U,N,{push,I,H},_)            ->
          {N1,H1} = chg(K-1,U,N-1,H,N-1), {N1,{push,I,H1}}.

repl(     pop, H={pop,_},_)            -> { 0,         H};
repl({push,I},{push,_,H},_)            -> { 0,{push,I,H}};
repl({push,I},   {pop,H},_)            -> { 2,{push,I,H}};
repl(     pop,{push,_,H},M) when M > 1 -> {-2,   {pop,H}}.
```

Notice how this change of data structure has an impact on the maintenance of `last/3` because we cannot anymore match any update with an underscore, thus the new version of this function is longer.

Persistent associative arrays. Let us explore the possibilities offered by the persistence of purely functional data structures by implementing a *persistent associative array*. A traditional array is a contiguous chunk of memory which can be accessed randomly by means of integer indexes ranging over an interval. An array has its length fixed at its creation and contains as many values as indexes are allowed. Associative arrays are commonly found in scripting languages, where they generalise arrays by accepting indexes to be any kind of value, not just integers. As a consequence, the notion of interval of indexes is dropped. All these arrays have in common the fact that they are updated destructively, that is, the value associated to some index, once updated, is lost. Arrays of both kinds are a staple feature of *imperative programming*. To obtain a persistent associative array, our first attempt will make use of the update-based persistence we studied above. Let us suppose that we only have two updates on arrays:

- `set` records an assignment of a given value at a given index, for instance, `{set,d,5}` specifies that the value at index `d` is `5` and any previous value at that index is now part of an old version;

- `unset` records the cancellation of a previous `set` update at a given index, for instance, `{unset,d}` means to unset any previous `set` update at index `d`. If the index was not in use or was already unset, `unset` has not effect, otherwise the previous value at that index becomes the latest, that is, it becomes part of the current version (if constructed).

A history is made of a list of such updates, starting with the empty list, which is interpreted as an empty array. Notice that, in contrast with the histories we dealt with previously, *we record here the `unset` update itself.* Here, it is not allowed to change the past directly: the only way to cancel a previous `set` update is to add an `unset` update to the history—history grows monotonically. Of course, sometimes a version of the array needs to be built, so a list of pairs index-value of the latest assigned indexes will be constructed from the history. This kind of list is called an *association list* and the indexes are then often called *keys* in this context. The reason why we started with the denomination "associative array" is because we wanted to provide the same functionalities as an array, but using a list. Hence, the array can be considered as the specification here, whilst the list is the actual implementation. Note that, since in general keys are not required to be totally ordered, several up-to-date lists may correspond to one array. Let us consider two basic functions on these persistent associative arrays.

- The function `get/2` accesses a value by means of its given key, for instance, `get(d,`H`)` evaluates to the value associated with the key `d` in the history H. If the key was not last assigned by `set`, the atom `none` results. Notice that the key in this example is an atom, but any kind of value is permited.
- The function `ver/2` is such that `ver(`n`,`H`)` builds the version n steps before the present moment in history H, the current version corresponding to 0 steps.

Let us start with the former. We need to make five cases: one for the empty history and two for each update, depending on whether the sought key is present in the last update or not:

`get(_,             [])` $\xrightarrow{\alpha}$ `absent;`
`get(X,[{set,X,Y}|H])` $\xrightarrow{\beta}$ $\boxed{}$`;`   `% Index found`
`get(X,[{set,A,B}|H])` $\xrightarrow{\gamma}$ $\boxed{}$`;`
`get(X,[{unset,X}|H])` $\xrightarrow{\delta}$ $\boxed{}$`;`   `% Index found`
`get(X,[{unset,A}|H])` $\xrightarrow{\epsilon}$ $\boxed{}$`.`

Clause β corresponds to the case when we found the value associated with the key `X`: it is `Y`:

`get(X,[{set,X,Y}|_])` $\xrightarrow{\beta}$ `Y;`

Clause γ matches when we find an assignement about a different key, so we have to keep looking in `H`:

`get(X,[{set,_,_}|H])` $\xrightarrow{\gamma}$ `get(X,H);`

Clause ϵ finds a different key being unset, so this is not our concern and we also should pass our way:

`get(X,[{unset,_}|H])` $\xrightarrow{\epsilon}$ `get(X,H).`

The difficult one is clause δ because we found the key we are seeking to be unset. This means that looking forward in the history `H`, that is, the past, *the next assignment to the key `X` must be ignored*, and the search should resume. This sounds very much like a variant of `get/2`, we could call `skip/2`. Here are the final definitions:

```
get(_,          []) -> absent;
get(X,[{set,X,Y}|_]) -> Y;
get(X,[{set,_,_}|H]) -> get(X,H);
get(X,[{unset,X}|H]) -> skip(X,H);       % Skip the next set...
get(X,[{unset,_}|H]) -> get(X,H).

skip(_,          []) -> absent;
skip(X,[{set,X,_}|H]) -> get(X,H);
skip(X,[{set,_,_}|H]) -> skip(X,H);                 % ...done.
skip(X,[{unset,_}|H]) -> skip(X,H).
```

The usual improvement in terms of memory usage consists in getting rid of the list:

```
get(_,        {}) -> absent;
get(X,{set,X,Y,_}) -> Y;
get(X,{set,_,_,H}) -> get(X,H);
get(X,{unset,X,H}) -> skip(X,H);
get(X,{unset,_,H}) -> get(X,H).

skip(_,        {}) -> absent;
skip(X,{set,X,_,H}) -> get(X,H);
skip(X,{set,_,_,H}) -> skip(X,H);
skip(X,{unset,_,H}) -> skip(X,H).
```

Exercises. [See answers page 378.]

1. Consider the following variant of `ver/2`:

   ```
   ver(0,H) -> last([],rev(H)).
   ```

```
last(    V,          []) -> V;
last(    V,[{push,I}|R]) -> last([I|V],R);
last([_|V],     [pop|R]) -> last(V,R).

rev(L)            -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Find the delay and the number of pushes.

2. The function call `ver(`n`,`H`)` on page 180 builds the version at n steps in the history H counted from the present version. Define a function `mk_ver/2` such that `mk_ver(`n`,`H`)` is the version at position n *counted from the origin* of the history H. Find the delay.
3. Write a definition of `ver/2` for persistent associative arrays and compare it with the one on page 180.
4. Let us define a variation on the theme of the persistent array. We wish to grant the user the possibility to think an array to be a contiguous piece of memory whose basic components, called *cells*, are accessed through their *index*. Contrary to associative arrays, indexes must range over an integer interval, from a *lower bound* to an *upper bound*. For example, the following array is made of five cells containing a, b, c, d and e, which have the respective indexes 4, 5, 6, 7 and 8, the lower bound being 4 and the upper bound 8:

4	5	6	7	8
a	b	c	d	e

An empty array is implemented by an empty tuple. A non-empty array is a triple made of

(a) a pair of integers representing the index bounds;
(b) a default value for unassigned cells;
(c) a series of updates in the shape of nested tuples.

Contrary to persistent associative arrays, there is only one update, `{set,`E`,`I`,`A`}`, which represents the assignment of the value E at index I in the array A. It is possible to reassign a cell, that is, to specify two updates for the same index (this enables the interpretation of the data structure as an imperative array) and also to undo an update (which supposes the array to be persistent as well). The function `init/3` is such that `init(`L`,`U`,`D`)` creates an array whose indexes range from the lower bound L to the upper bound $U \geqslant L$ and containing the default value of D. We simply have

```
init(L,U,D) when L =< U -> {{L,U},D,{}}.
```

Define the following functions:

• `ver/1` is such that `ver(`A`)` is rewritten into an association list containing the bindings index-item lastly assigned in the array A. In particular, values of reassigned indexes are ignored. The order of the bindings is not significant.

• `set/3` is such that `set(`E`,`I`,`A`)` is an array obtained by assigning the value of E to the index I of the non-empty array A. If the index is out of bounds, including if the array is empty, then the result is the atom `out`.

• `get/2` is such that `get(`A`,`I`)` is the content of the cell in the array A at index I. If the index is out of bound, the atom `out` results.

• `nth/2` is such that `nth(`A`,`I`)` is the content of the Ith cell in the array A. If the intended index is out of bounds, the atom `out` results.

• `mem/2` is such that `mem(`A`,`E`)` rewrites to the last (temporally) index in the array A to have been assigned E. If the sought value has been reassigned, it is not part of the last version of the array, thus it is not found. If not found, the atom `absent` is the final value.

• `inv/1` is such that `inv(`A`)` evaluates to an array whose cell contents are in reverse order with respect to array A. The history of all updates must be preserved, only indexes change.

• `unset/2` is such that `unset(`A`,`I`)` returns an array identical to A except that the last assignment to the cell of index I is undone. If the cell in question was never assigned, the result is A. If the index is out of bounds, the atom `out` results. Note that `unset` was an update when we presented earlier the persistent associative arrays.

Chapter 9

Permutations and Sorting

Intimately connected to the issue of *sorting*, which we shall study later, is the problem of generating all *permutations* of a finite set of objects. Without any loss of generality and for the sake of simplicity, let us consider the tuple $(1, 2, \ldots, n)$. A permutation of this tuple is another tuple $(a_1, a_2, \ldots, a_n)$ such that each a_i is one of the integers, such that $a_i \neq a_j$ for all $i \neq j$. For example, all the permutations of $(1, 2, 3)$ are

$$(1, 2, 3) \quad (1, 3, 2) \quad (2, 1, 3) \quad (2, 3, 1) \quad (3, 1, 2) \quad (3, 2, 1).$$

How many permutations of a group of n objects are there? This question is easy to answer by considering the components in the tuple from left to right. The first component can be filled with an object amongst n. For the second, we can choose only amongst $n - 1$, since we already picked an object for the first component and we are not allowed to repeat it. Finally, the last component is compulsorily the remaining object. Since choosing an object for each component does not depend on the previous selections (only non-repetition is disallowed), the number of possible choices is multiplied each time, that is, there are $n \cdot (n - 1) \cdot \ldots \cdot 1 = n!$ distinct permutations. In the example above, we indeed counted $3! = 6$ permutations of $(1, 2, 3)$. This is not the first time we have encountered the factorial function. Here is a simple derivation enabling the characterisation of its asymptotic growth, proposed by Graham, Knuth and Patashnik in their famous textbook *Concrete Mathematics* (Addison-Wesley). We start by squaring the factorial and regrouping the factors as follows:

$$n!^2 = (1 \cdot 2 \cdot \ldots \cdot n)(n \cdot \ldots \cdot 2 \cdot 1) = \prod_{k=1}^{n} k(n + 1 - k).$$

The parabola $P(k) := k(n + 1 - k) = -k^2 + (n + 1)k$ reaches its maximum where its derivative is zero: $P'(k_{\max}) = 0 \Leftrightarrow k_{\max} = (n + 1)/2$. The corresponding ordinate is $P(k_{\max}) = ((n + 1)/2)^2 = k_{\max}^2$.

When k ranges from 1 to n, the minimal ordinate, n, is reached at absissas 1 and n, as shown in FIGURE 54 on page 196. Hence, $1 \leqslant k \leqslant k_{\max}$ implies

$$P(1) \leqslant P(k) \leqslant P(k_{\max}), \quad \text{that is,} \quad n \leqslant k(n+1-k) \leqslant \left(\frac{n+1}{2}\right)^2.$$

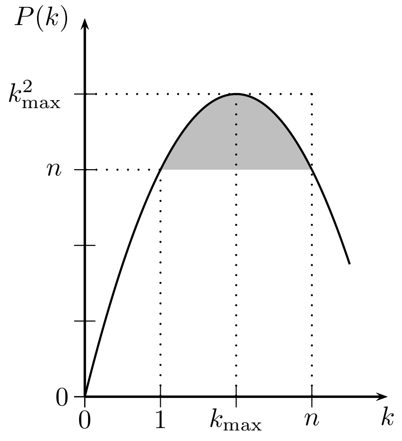


FIGURE 54: Parabola $P(k) := k(n+1-k)$

Multiplying the sides by varying k over the discrete interval $[1..n]$ yields

$$n^n = \prod_{k=1}^{n} n \leqslant n!^2 \leqslant \prod_{k=1}^{n} \left(\frac{n+1}{2}\right)^2 = \left(\frac{n+1}{2}\right)^{2n} \Rightarrow n^{n/2} \leqslant n! \leqslant \left(\frac{n+1}{2}\right)^n$$

It is clear now that $n!$ is *exponential*, so it asymptotically outgrows any polynomial. Concretely, a function whose delay is proportional to a factorial is useless even for small inputs. For the cases where an equivalence is preferred, Stirling's formula states that

$$n! \sim n^n e^{-n}\sqrt{2\pi n}, \quad \text{as } n \rightarrow \infty.$$

The reckoning we used to count all permutations of $(1,2,\ldots,n)$ makes use of series of choices among a set of decreasing size. Instead of thinking globally like this, which is often a daunting task, let us instead try inductively, that is, let us find a way, given all the permutations of $(1,2,\ldots,n-1)$, to build all the permutations of $(1,2,\ldots,n)$. If $(a_1,a_2,\ldots,a_{n-1})$ is a permutation of $(1,2,\ldots,n-1)$, then we can construct n permutations of $(1,2,\ldots,n)$ by inserting the number n at all possible places in $(a_1,a_2,\ldots,a_{n-1})$: before a_1, between a_2 and a_3, etc., until after a_{n-1}:

$(\boldsymbol{n}, a_1, a_2, \ldots, a_{n-1})$ $(a_1, \boldsymbol{n}, a_2, \ldots, a_{n-1})$ $\ldots$ $(a_1, a_2, \ldots, a_{n-1}, \boldsymbol{n})$.

For example, it is obvious that all the permutations of $(1, 2)$ are $(1, 2)$ and $(2, 1)$. Our method leads from $(1, 2)$ to

$$(\mathbf{3}, 1, 2) \quad (1, \mathbf{3}, 2) \quad (1, 2, \mathbf{3})$$

and from $(2, 1)$ to

$$(\mathbf{3}, 2, 1) \quad (2, \mathbf{3}, 1) \quad (2, 1, \mathbf{3}).$$

If we name p_n the number of permutations on n elements, we draw from this the recurrence $p_n = n \cdot p_{n-1}$, which, with the additional obvious $p_1 = 1$, leads to $p_n = n!$, for all $n > 0$, exactly as expected. If the n objects to permute are not $(1, 2, \ldots, n)$, for example, $(\mathsf{b}, \mathsf{d}, \mathsf{a}, \mathsf{c})$, simply associate each of them to their index in the tuple, for example, `b` is represented by 1, `d` by 2, `a` by 3 and `c` by 4, so the tuple is then associated to $(1, 2, 3, 4)$ and, for instance, the permutation $(4, 1, 2, 3)$ means $(\mathsf{c}, \mathsf{b}, \mathsf{d}, \mathsf{a})$.

Let us start by defining the function `perm/1` such that the call `perm(`L`)` is the list of all permutations of the items of list L. We implement the inductive method presented above, which worked by inserting a new object into all possible places of a shorter permutation. We have

```
perm(   []) -α-> [];
perm(  [I]) -β-> [[I]];
perm([I|L]) -γ-> dist(I,perm(L)).
```

where the function `dist/2` ("distribute") is such that a call `dist(`I`,`L`)` inserts the item I into all places of all the permutations contained in the list L. Because the insertion of I everywhere into a given permutation of length n leads to $n+1$ permutations of length $n+1$, we must append these new permutations to the others of same length:

```
dist(_,          []) -δ-> [];
dist(I,[Perm|Perms]) -ε-> join(ins(I,Perm),dist(I,Perms)).
```

The call `ins(I,Perm)` computes the list of permutations resulting from inserting `I` in all places in the permutation `Perm`. We thus derive

```
ins(I,        []) -ζ-> [[I]];
ins(I,Perm=[J|L]) -η-> [[I|Perm]|push(J,ins(I,L))].
```

where the function `push/2` is such that any call `push(`I`,`L`)` pushes item I on top of all the permutations of the list of permutations L. The order, as usual up to now, is unchanged:

```
push(_,          []) -θ-> [];
push(I,[Perm|Perms]) -ι-> [[I|Perm]|push(I,Perms)].
```

Now we can compute all the permutations of $(4, 1, 2, 3)$ or $(\mathtt{c}, \mathtt{b}, \mathtt{d}, \mathtt{a})$ by calling `perm([4,1,2,3])` or `perm([c,b,d,a])`. Note that, after computing the permutations of length $n + 1$, the permutations of length n are not needed anymore, which allows the garbage collector to reclaim the corresponding memory cells for other uses. As far as the delays are concerned, the definition of `push/2` leads to

$$\mathcal{D}_0^{\mathtt{push}} \overset{\theta}{=} 1; \qquad \mathcal{D}_{n+1}^{\mathtt{push}} \overset{\iota}{=} 1 + \mathcal{D}_n^{\mathtt{push}}, \quad \text{with } n \geqslant 0.$$

We can easily deduce a simple closed expressions for the delay:

$$\mathcal{D}_n^{\mathtt{push}} = n + 1, \quad \text{with } n \geqslant 0.$$

We know that the result of `ins(`I`,`π`)` is a list of length $n + 1$ if π is a permutation of n objects into which we insert one more object I. Hence, the definition of `ins/2` leads to

$$\mathcal{D}_0^{\mathtt{ins}} \overset{\zeta}{=} 1; \quad \mathcal{D}_{p+1}^{\mathtt{ins}} \overset{\eta}{=} 1 + \mathcal{D}_{p+1}^{\mathtt{push}} + \mathcal{D}_p^{\mathtt{ins}} = 3 + p + \mathcal{D}_p^{\mathtt{ins}}, \quad \text{with } p \geqslant 0,$$

where $\mathcal{D}_p^{\mathtt{ins}}$ is the delay of `ins(`I`,`π`)` with π of length p. By summing for all p from 0 to $n - 1$, for $n > 0$, on both sides and then cancelling equals, we have

$$\sum_{p=0}^{n-1} \mathcal{D}_{p+1}^{\mathtt{ins}} = \sum_{p=0}^{n-1} 3 + \sum_{p=0}^{n-1} p + \sum_{p=0}^{n-1} \mathcal{D}_p^{\mathtt{ins}},$$

By cancelling identical terms in the sums $\sum_{p=0}^{n-1} \mathcal{D}_{p+1}^{\mathtt{ins}}$ and $\sum_{p=0}^{n-1} \mathcal{D}_p^{\mathtt{ins}}$ (this is called the *telescoping* or *difference* method), we draw

$$\begin{aligned}\mathcal{D}_n^{\mathtt{ins}} &= 3n + n(n-1)/2 + \mathcal{D}_0^{\mathtt{ins}} = 3n + n(n-1)/2 + 1, \quad \text{by } (\overset{\zeta}{=}),\\ &= \frac{1}{2}n^2 + \frac{5}{2}n + 1.\end{aligned}$$

This last equation is actually valid even if $n = 0$. Let $\mathcal{D}_{n!}^{\mathtt{dist}}$ be the delay for distributing an item among $n!$ permutations of length n. The definition of `dist/2` shows that it repeats calls to `join/2` and `ins/2` whose arguments are always of length $n+1$ and n, respectively, because all processed permutations here have the same length. Thus, we deduce, for $p \geqslant 0$, that

$$\mathcal{D}_0^{\mathtt{dist}} \overset{\delta}{=} 1; \quad \mathcal{D}_{p+1}^{\mathtt{dist}} \overset{\epsilon}{=} 1 + \mathcal{D}_{n+1}^{\mathtt{join}} + \mathcal{D}_n^{\mathtt{ins}} + \mathcal{D}_p^{\mathtt{dist}} = \frac{1}{2}n^2 + \frac{7}{2}n + 4 + \mathcal{D}_p^{\mathtt{dist}},$$

since we already know that $\mathcal{D}_n^{\mathtt{join}} = n + 1$ and the value of $\mathcal{D}_n^{\mathtt{ins}}$. By summing both sides of the last equation for all p from 0 to $n! - 1$, we can eliminate most of the terms and find a non recursive definition of

$\mathcal{D}_{n!}^{\texttt{dist}}$, for $n > 0$:

$$\sum_{p=0}^{n!-1} \mathcal{D}_{p+1}^{\texttt{dist}} = \sum_{p=0}^{n!-1} \left(\frac{1}{2}n^2 + \frac{7}{2}n + 4\right) + \sum_{p=0}^{n!-1} \mathcal{D}_{p}^{\texttt{dist}},$$

$$\mathcal{D}_{n!}^{\texttt{dist}} = \left(\frac{1}{2}n^2 + \frac{7}{2}n + 4\right) n! + \mathcal{D}_{0}^{\texttt{dist}} = (n^2 + 7n + 8)\frac{n!}{2} + 1.$$

Let us finally compute the delay of `perm(`L`)`, noted $\mathcal{D}_{p}^{\texttt{perm}}$, where p is the length of the list L. From the clauses α to γ, we deduce the following equations, where $p > 0$:

$$\mathcal{D}_{0}^{\texttt{perm}} \overset{\alpha}{=} 1; \qquad \mathcal{D}_{1}^{\texttt{perm}} \overset{\beta}{=} 1;$$
$$\mathcal{D}_{p+1}^{\texttt{perm}} \overset{\gamma}{=} 1 + \mathcal{D}_{p}^{\texttt{perm}} + \mathcal{D}_{p!}^{\texttt{dist}} = (p^2 + 7p + 8)p!/2 + 2 + \mathcal{D}_{p}^{\texttt{perm}}.$$

Again, summing both sides, most of the terms cancel out:

$$\begin{aligned}
\sum_{p=1}^{n-1} \mathcal{D}_{p+1}^{\texttt{perm}} &= \frac{1}{2}\sum_{p=1}^{n-1} (p^2 + 7p + 8)p! + \sum_{p=1}^{n-1} 2 + \sum_{p=1}^{n-1} \mathcal{D}_{p}^{\texttt{perm}} \\
\mathcal{D}_{n}^{\texttt{perm}} &= \frac{1}{2}\sum_{p=1}^{n-1} (p^2 + 7p + 8)p! + 2(n-1) + \mathcal{D}_{1}^{\texttt{perm}}, \text{ with } n > 1, \\
&= \frac{1}{2}\sum_{p=1}^{n-1} ((p+2)(p+1) + 6 + 4p)p! + 2n - 1 \\
&= \frac{1}{2}\sum_{p=1}^{n-1} (p+2)(p+1)p! + 3\sum_{p=1}^{n-1} p! + 2\sum_{p=1}^{n-1} pp! + 2n - 1 \\
&= \frac{1}{2}\sum_{p=1}^{n-1} (p+2)! + 3\sum_{p=1}^{n-1} p! + 2\sum_{p=1}^{n-1} pp! + 2n - 1 \\
&= \frac{1}{2}\sum_{p=3}^{n+1} p! + 3\sum_{p=1}^{n-1} p! + 2\sum_{p=1}^{n-1} pp! + 2n - 1 \\
&= \left(\frac{1}{2}((n+1)! + n! - 2! - 1!) + \frac{1}{2}\sum_{p=1}^{n-1} p!\right) + 3\sum_{p=1}^{n-1} p! \\
&\quad + 2\sum_{p=1}^{n-1} pp! + 2n - 1. \\
&= \frac{1}{2}(n+2)n! + \frac{7}{2}\sum_{p=1}^{n-1} p! + 2\sum_{p=1}^{n-1} pp! + 2n - \frac{5}{2}.
\end{aligned}$$

This last equation is actually valid even if $n = 1$. Let us prove by induction the conjecture

$$\sum_{p=1}^{n-1} pp! = n! - 1, \quad \text{for } n > 2.$$

First, we check the equation for the smallest value of n:

$$\sum_{p=1}^{1-1} pp! := 0 = 1! - 1.$$

Now, let us suppose the conjecture valid for a given n, called the *induction hypothesis*, and let us prove that it holds for $n + 1$:

$$\sum_{p=1}^{n} pp! = \sum_{p=1}^{n-1} pp! + nn!.$$

We can use the induction hypothesis now:

$$\sum_{p=1}^{n} pp! = (n! - 1) + nn! = (n + 1)n! - 1 = (n + 1)! - 1,$$

which we recognise as the equality to be proved for $n + 1$, hence the theorem is valid for all $n > 0$, according to the induction principle. For such values of n, it then leads to

$$\begin{aligned}\mathcal{D}_n^{\mathsf{perm}} &= \frac{1}{2}nn! + n! + \frac{7}{2}\sum_{p=1}^{n-1} p! + 2(n! - 1) + 2n - \frac{5}{2}\\ &= \frac{1}{2}nn! + 3n! + 2n - \frac{9}{2} + \frac{7}{2}\sum_{p=1}^{n-1} p!, \quad \text{with } n > 0.\end{aligned}$$

The remaining sum is called the *left factorial* and is usually defined

$$!n := \sum_{p=1}^{n-1} p!, \quad \text{with } n > 0.$$

Unfortunately, no closed expression of the left factorial is known. This is actually a common situation when determining the delay of relatively complex Erlang functions. When confronted to this kind of situation, the best course of action is to study the asymptotic approximation of the delay. First, we remark that

$$\begin{aligned} !n &= (n - 1)! + !(n - 1),\\ !n/(n - 1)! &= 1 + !(n - 1)/(n - 1)!\\ &= 1 + ((n - 2)! + !(n - 2))/(n - 1)!\end{aligned}$$

$$\begin{aligned} &= 1 + 1/(n-1) + !(n-2)/(n-1)! \\ !n/(n-1)! &= 1 + 1/(n-1) + 1/(n-1)(n-2) + \cdots + 1/(n-1)!. \end{aligned}$$

Furthermore, the following obvious inequalities hold for $n > 0$:

$$\begin{aligned} 1 &\leqslant 1, \\ n-1 &\leqslant n-1, \\ n-1 &< (n-1)(n-2), \\ &\vdots \\ n-1 &< (n-1)(n-2)\cdot\ldots\cdot 1 = (n-1)!. \end{aligned}$$

Inversing the sides and summing them we deduce

$$\begin{aligned} 1 + \frac{n-1}{n-1} &> 1 + \frac{1}{n-1} + \frac{1}{(n-1)(n-2)} + \cdots + \frac{1}{(n-1)!}, \\ 2 &> \frac{!n}{(n-1)!}. \end{aligned}$$

Therefore, we have the bounds

$$(n-1)! < !n < 2(n-1)!.$$

This is enough to proceed conclusively. Let us define the bounds for $\mathcal{D}_n^{\mathsf{perm}}$ as $L(n) < \mathcal{D}_n^{\mathsf{perm}} < U(n)$, where

$$\begin{aligned} L(n) &:= \frac{1}{2}nn! + 3n! + 2n - \frac{9}{2} + \frac{7}{2}(n-1)!, \\ U(n) &:= \frac{1}{2}nn! + 3n! + 2n - \frac{9}{2} + 7(n-1)!. \end{aligned}$$

Then

$$\begin{aligned} \frac{2}{nn!}L(n) &= 1 + \frac{6}{n} + \frac{4}{n!} - \frac{9}{nn!} + \frac{7}{n^2} \rightarrow 1, \quad \text{as } n \rightarrow \infty, \\ \frac{2}{nn!}U(n) &= 1 + \frac{6}{n} + \frac{4}{n!} - \frac{9}{nn!} + \frac{14}{n^2} \rightarrow 1, \quad \text{as } n \rightarrow \infty, \end{aligned}$$

that is, $L(n) \sim U(n) \sim \frac{1}{2}nn!$. Hence

$$\mathcal{D}_n^{\mathsf{perm}} \sim \frac{1}{2}nn!, \quad \text{as } n \rightarrow \infty.$$

Stirling's approximation allows us finally to conclude

$$\mathcal{D}_n^{\mathsf{perm}} \sim \sqrt{\frac{\pi n}{2}}e^{-n}n^{n+1}, \quad \text{as } n \rightarrow \infty.$$

This is an unbearably slow program, as expected. We should not hope to compute $\mathcal{D}_{11}^{\mathsf{perm}}$ easily and there is no way to improve significantly the delay because the number of permutations it computes is inherently exponential, so it would even suffice to spend only one function call per permutation to obtain an exponential delay.

Permutations are worth studying in detail because of their intimate relationship with *sorting*, which is the process of ordering a finite series of objects. The result of a sort can indeed be viewed as a permutation of the objects of a given permutation, so that they become ordered. In other words, the initial permutation can be thought of as scrambling originally ordered objects and the sorting permutation then puts them back to their place. To understand this precisely, it is perhaps helpful to use a slightly different notation for permutations, one which shows the indexes together with the objects. For example, instead of writing unidimensionally $\pi_1 = (2, 4, 1, 5, 3)$, we would write

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \end{pmatrix}.$$

The first line is made of the ordered indexes and the second line contains the objects, which are integers in the same range as the indexes, usually unordered. In general, if we have a permutation $\pi = (a_1, a_2, \ldots, a_n)$, it is written with more details as

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix},$$

where $a_i = \pi(i)$, for all i from 1 to n. For the sake of simplicity, we shall require that the objects are the same as the indexes, that is, integers in the same interval starting at 1. Then the result of any sorting algorithm on π_1 is the following permutation π_s, which has to be interpreted as a permutation on the objects of π_1:

$$\pi_s = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 2 & 4 \end{pmatrix}.$$

Technically, to see how it works, we need first to define the *composition of two permutations* π_a and π_b as

$$\pi_b \circ \pi_a := \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi_b(\pi_a(1)) & \pi_b(\pi_a(2)) & \ldots & \pi_b(\pi_a(n)) \end{pmatrix},$$

then the permutation π_s sorting π_1 has the defining property that $\pi_s \circ \pi_1 = \mathcal{I}$, where the *identity permutation* $\mathcal{I}$ is such that $\mathcal{I}(i) = i$, for all indexes i. In other words, $\pi_s = \pi_1^{-1}$, that is, *sorting a permutation consists in building its inverse.* If one considers a permutation, other than the identity, to shuffle initially ordered numbers, its inverse puts them back to their original place. We can now check that, indeed,

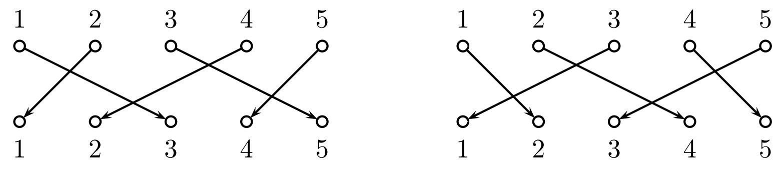

(a) Bigraph of $\pi_1^{-1} = (3, 1, 5, 2, 4)$ (b) Bigraph of $\pi_1 = (2, 4, 1, 5, 3)$

FIGURE 55: Sorting permutation π_1^{-1} and permutation to be sorted π_1

$\pi_1^{-1} \circ \pi_1 = \mathcal{I}$:

$$\begin{pmatrix}1 & 2 & 3 & 4 & 5\\ 3 & 1 & 5 & 2 & 4\end{pmatrix} \circ \begin{pmatrix}1 & 2 & 3 & 4 & 5\\ 2 & 4 & 1 & 5 & 3\end{pmatrix} = \begin{pmatrix}1 & 2 & 3 & 4 & 5\\ 1 & 2 & 3 & 4 & 5\end{pmatrix}.$$

An alternative representation of permutations and their composition is based on considering them as *bijections from an interval onto itself*, denoted by *bipartite graphs*, also called *bigraphs*. Such graphs are made of two disjoint, ordered sets of vertices of same cardinal, the indexes and the objects, and the edges always go from an index to an object, without sharing the vertices with other edges. Note that we state nothing about the nature of the indexes and the objects, as only the total order over both and their relationships do matter, but, for consistency with the previous presentation, we shall assume that both the indexes and the objects are integers belonging to the same interval. For example, permutation π_1 is shown in FIGURE 55b on page 203 and its inverse, π_1^{-1}, is displayed in FIGURE 55a on page 203. The composition of π_1^{-1} and π_1 is then obtained by identifying the object vertices of π_1 with the index vertices of π_1^{-1}, as shown in FIGURE 56a on page 203. The resulting identity permutation, $\mathcal{I}$, is obtained by replacing two adjacent edges by their transitive closure and erasing the intermediate vertices, as shown in FIGURE 56b on page 203. Note that the edges in $\mathcal{I}$

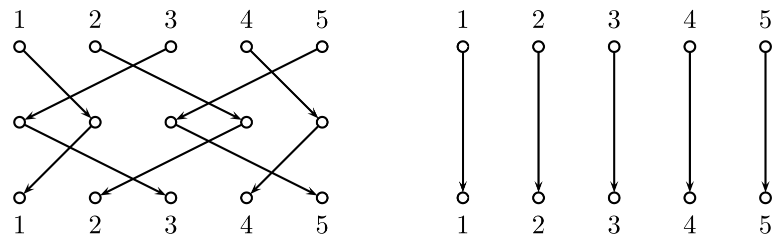

(a) $\pi_1^{-1} \circ \pi_1$ (b) $\mathcal{I} = \pi_1^{-1} \circ \pi_1$

FIGURE 56: Applying π_1 to its sorting permutation π_1^{-1}.

do not cross over each other: it is a characteristic feature of the identity permutation. As an amusing occurrence, it may be that $\pi_1^{-1} = \pi_1$, that is, the given permutation is equal to its inverse, or, in other words, the sorting permutation may be the same as the permutation to be sorted. Consider for instance the permutation

$$\pi_3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 2 & 5 \end{pmatrix}$$

and, in FIGURE 57 on page 204, the representation as a bigraph of $\pi_3 \circ \pi_3$. This kind of permutation, such that $\pi_3 \circ \pi_3 = \mathcal{I}$, is called an *involution*.

Studying permutations and their basic properties helps understanding sorting algorithms, in particular, their average delay. It also provides a way to quantify the disorder of a permutation. Given $(1, 3, 5, 2, 4)$, we can see that only the pairs of objects $(3, 2)$, $(5, 2)$ and $(5, 4)$ are out of order. In general, given $(a_1, a_2, \ldots, a_n)$, the pairs (a_i, a_j) such that $i < j$ and $a_i > a_j$ are called *inversions*. The more inversions, the greater disorder. As expected, the identity permutation has no inversions and previously studied permutation $\pi_1 = (2, 4, 1, 5, 3)$ has 4. When considering permutations as represented by bigraphs, an inversion corresponds to an intersection of two edges, more precisely, it is the pair made of the objects pointed at by two arrows. Therefore, the number of inversions corresponds to the number of edge crossings, so, for instance, π_1^{-1} has 4 inversions. In fact, *the inverse of a permutation has the same number of inversions as the permutation itself.* This can be clearly seen when comparing the digraphs of π_1 and π_1^{-1} in FIGURE 55: in order to deduce the bigraph of π_1^{-1} from the one corresponding to π_1, let us reverse each edge, that is, the direction in which the arrows are pointing, then swap the indexes and the objects, that is, exchange the two lines of vertices—alternatively, we can imagine that we fold down the paper along the object line, then look through and reverse the arrows. The horizontal symmetry is especially obvious in FIGURE 56a and FIGURE 57a on the following page. From a programming standpoint, the sorting problem is stated as follows: given a list of integers, we want to order them in *nondecreasing order*, that is, to output a list containing the same numbers with the additional property that, for all integers N and M, if N is located before M, then $N \leqslant M$. For example, if the input list is `[6,3,1,7,5,3,0]`, the result is `[0,1,3,3,5,6,7]`. Note that equal integers will end up being adjacent and they are the reason why we qualified the sort as non-decreasing, instead of increasing—the latter assuming strict monotonicity. This programming approach to sorting can be understood if permutations are kept in mind, although there

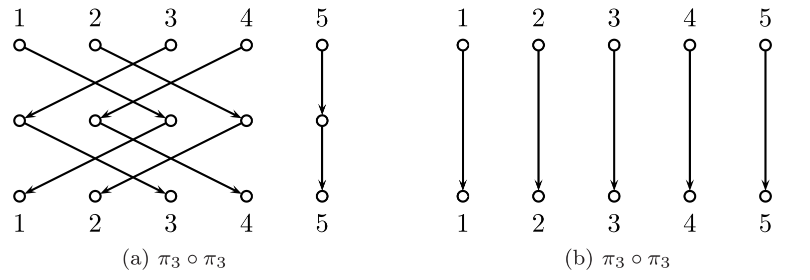


FIGURE 57: Involution π_3 sorts itself

are slight practical issues, which are threefold. First, the objects may not necessarily be integers in the same range as the indexes. This is not a real obstacle because what matters most in a permutation is the relative order of the objects, irrespective of their nature. Second, the result as just stated is not the sorting permutation, but the sorted permutation, which corresponds to the identity permutation in our theoretical presentation above. In fact, we are not going to build any sorting permutation explicitly, but, instead, the sorted permutation directly. Last, we allow the permutation objects to be repeated, that is, they do not constitute a mathematical set. This situation can be handled by considering that an object appearing before another equal object is actually smaller, as far as the ordering relationship is concerned. Take for instance the permutation $(\mathsf{d}, \mathsf{b}, \mathsf{a}, \mathsf{a}, \mathsf{c})$, where `a` occurs twice. We then decide, during the sorting process, that the first occurrence is smaller than the second one. One may wonder if that is an issue at all, since there is no way to distinguish between the two objects `a`, but later on we shall attach potentially different data to these `a`, so they will be considered different in certain contexts. A sorting algorithm that thus maintain the initial order between identical objects is called *stable*, otherwise it is said to be *unstable*.

Exercise. [See answer page 380.]
Transform `perm/1` and all the definitions it depends upon into tail form.

Chapter 10

Insertion Sort

Let us start by the following observation. At any time, if we already have a sorted list, it is easy to insert one more number at the right place by comparing it with the first number, then, if necessary, the second, the third etc. If it is greater than all the numbers in the list, we must build a list where the number is present at the end. For example, inserting `1` in `[3,6]` requires comparing `1` and `3` and, since `1 =< 3`, the result is `[`**`1`**`,3,6]`—we don't need to compare `1` to `6`. When inserting `4` in `[1,3,6]`, we compare first `4` and `1`; because `4 > 1`, we must compare `4` with `3`; since `4 > 3`, we must further compare `4` and `6`; because `4 =< 6`, we know that the result is `[1,3,`**`4`**`,6]`. Inserting `7` in `[3,6]` results in comparing `7` with `3` and `6`, then concluding `[3,6,`**`7`**`]`. Inserting a number in the empty list results in no comparison and the result is a singleton list. The algorithm, called *insertion sort*, consists then in inserting the numbers one by one in a list originally empty.

One-way insertion sort. The previous description shows that insertion sort requires two functions: one to insert a number in a sorted list and another to traverse the input and call the insertion function for each number encountered. Let us focus first on `insert/2`, such that `insert(`I`,`S`)` is rewritten in the sorted list containing all the numbers in the sorted list S plus I. As usual, we reason by case analysis and prepare patterns matching the empty list and the non-empty list:

```
insert(I,   []) -> □;
insert(I,[J|S]) -> □.
```

The first clause is easy to complete: the result is the singleton list `[I]` and no comparison is involved (we have only one number at hand, anyway):

```
insert(I,   []) -> [I];
insert(I,[J|S]) -> □.
```

The second clause must be split because either `I =< J` or `I > J`. Let us try

```
insert(I,   [])             -> [I];
insert(I,[J|S]) when I =< J -> □;
insert(I,[J|S])             -> □.
```

If `I =< J`, then `I` must be before `J` in the output, therefore we can, here too, conclude in one step:

```
insert(I,   [])             -> [I];
insert(I,[J|S]) when I =< J -> [I,J|S];
insert(I,[J|S])             -> □.
```

Finally, if `I > J`, we know that `I` must be *after* `J` in the output, so we must keep `J` at its current relative position and insert `I` in `S`:

```
insert(I,   [])             -> [I];
insert(I,[J|S]) when I =< J -> [I,J|S];
insert(I,[J|S])             -> [J|insert(I,S)].
```

Focusing back on the second clause, we realise that we do nothing with `S` and we copy as a whole `[J|S]`, because `[I,J|S]` is just the same as `[I|[J|S]]`. As we saw earlier, this results in the creation of a node (`|`), while it would be more economical to refer to the node (`|`) in `[J|S]`. This is where an *alias* comes handy:

```
insert(I,   [])               -> [I];
insert(I,L=[J|S]) when I =< J -> [I|L];
insert(I,[J|S])               -> [J|insert(I,S)].
```

We could go further and silence `S` in the aliased pattern:

```
insert(I,   [])               -> [I];
insert(I,L=[J|_]) when I =< J -> [I|L];
insert(I,[J|S])               -> [J|insert(I,S)].
```

But this improvement is not entirely satisfying. What if we test `I > J` first? This results in swapping the two last clauses:

```
insert(I,     [])           -> [I];
insert(I,  [J|S]) when I > J -> [J|insert(I,S)];
insert(I,L=[J|_])            -> [I|L].
```

Nothing is done with `J` in the last clause, so we can silence it and rename `L` into `S` to remind the reader that it is a sorted list:

```
insert(I,   [])            $\xrightarrow{\gamma}$ [I];
insert(I,[J|S]) when I > J $\xrightarrow{\delta}$ [J|insert(I,S)];
insert(I,    S)            $\xrightarrow{\epsilon}$ [I|S].
```

This is better because we use only two variables instead of three and sharing is maximum without the need of an alias. Still, another look reveals that clause γ is useless if `S` in clause ϵ is allowed to match the empty list. The final word seems to be

```
insert(I,[J|S]) when I > J $\xrightarrow{\delta}$ [J|insert(I,S)];
insert(I,    S)            $\xrightarrow{\epsilon}$ [I|S].
```

The remaining function is `isort/1`, such that `isort(`L`)` is rewritten into the sorted list corresponding to list L. The only task left consists in walking the input list and inserting each number, that is, calling `insert/2`:

```
isort(   []) $\xrightarrow{\beta}$ [];
isort([I|L]) $\xrightarrow{\gamma}$ insert(I,isort(L)).
```

Let us compute now the delay. At this point it is worth mentioning that, traditionally, textbooks about program analysis assess the delays of sorting algorithms by counting the number of comparisons, not the number of function calls. Doing so allows one to compare with the same measure different sorting algorithms, as long as they perform comparisons. (There are sorting algorithms which do not rely on comparisons.) A glimpse at the definition of `insert/2` convinces us that the delay depends on the result of *all the comparisons* needed to insert a number. It is not possible to capture this into a single, exact number, therefore we have instead to think in terms of best, worst and average delays. How can we pick and arrange n integers into a list L which leads to the minimum delay for evaluating `isort(`L`)`? The first hint comes from the observation that such an input would not exert clause δ, since it is recursive, whereas it would rely on clause ϵ, which contains no call at all. In other words, each number to be inserted would be lower or equal than the head of the current sorted list. The second hint is that *the items are inserted in reverse order*, due to clause γ, that is, the last item is first inserted in the empty list, the penultimate is then inserted, thus compared to the last number in the input list (which is the first now). Since it is expected here to be lower, it means that the penultimate number in the input is lower than the last. Plainly, this means that *the best case happens when the input list is already nondecreasingly sorted.*

Let us note $\mathcal{B}_n^{\texttt{isort}}$ the delay in the best case for n items. Then the execution trace is $\gamma^n\beta\epsilon^n$. In total, $\mathcal{B}_n^{\texttt{isort}} = 2n + 1$.

The worst case must exert clause δ as much as possible, which implies that *the worst case happens when the input is a list decreasingly sorted.* Note the precision of the vocabulary: "decreasingly" excludes the possibility of two numbers being equal, contrary to "non-increasingly," which allows that to happen. Let us note $\mathcal{W}_n^{\mathsf{isort}}$ the delay in the worst case of n items to be sorted. It requires more care as the number of times clause δ is used is the length of its second argument, which means that we cannot count separately the number of rewrites by clauses γ and δ. In this case, it is wiser to write down the recurrence equations corresponding to the program, the measure being the length of the list. Assuming $p \geqslant 0$, we deduce that

$$\begin{aligned} &\mathcal{W}_0^{\mathsf{insert}} \overset{\epsilon}{=} 1; && \mathcal{W}_0^{\mathsf{isort}} \overset{\beta}{=} 1; \\ &\mathcal{W}_{p+1}^{\mathsf{insert}} \overset{\delta}{=} 1 + \mathcal{W}_p^{\mathsf{insert}}; && \mathcal{W}_{p+1}^{\mathsf{isort}} \overset{\gamma}{=} 1 + \mathcal{W}_p^{\mathsf{insert}} + \mathcal{W}_p^{\mathsf{isort}}. \end{aligned}$$

In equation ($\overset{\gamma}{=}$), the term $\mathcal{W}_p^{\mathsf{insert}}$ is correct because the call `isort(L)` evaluates in a list of the same length as `L`, that is, p. Notice that we use clause ϵ only in the case of the empty list, because, otherwise, we would use clause δ. It is now easy to deduce from equations ($\overset{\delta}{=}$) and ($\overset{\epsilon}{=}$) a closed expression for $\mathcal{W}_n^{\mathsf{insert}}$, where n is the length of the input list:

$$\mathcal{W}_n^{\mathsf{insert}} = n + 1, \text{ with } n \geqslant 0.$$

Replacing this expression in the remaining equations leads to

$$\mathcal{W}_0^{\mathsf{isort}} \overset{\beta}{=} 1, \quad \mathcal{W}_{p+1}^{\mathsf{isort}} \overset{\gamma}{=} 2 + p + \mathcal{W}_p^{\mathsf{isort}}.$$

Summing both sides of ($\overset{\gamma}{=}$) and simplifying (this is the telescoping technique), we get

$$\begin{aligned} \sum_{p=0}^{n-1} \mathcal{W}_{p+1}^{\mathsf{isort}} &= \sum_{p=0}^{n-1} 2 + \sum_{p=0}^{n-1} p + \sum_{n=0}^{n-1} \mathcal{W}_p^{\mathsf{isort}}, \\ \mathcal{W}_n^{\mathsf{isort}} &= 2n + n(n-1)/2 + \mathcal{W}_0^{\mathsf{isort}} = (n^2 + 3n + 2)/2 \\ &= \frac{1}{2}(n+1)(n+2) = \sum_{k=1}^{n+1} k \sim \frac{1}{2}n^2, \text{ as } n \rightarrow \infty. \end{aligned}$$

In short, `isort/1` has a quadratic delay in the worst case, which is when the input is sorted in decreasing order.

There is another way to find the result, by means of the execution trace T_w of the worst case. We expect it to be

$$T_w := \gamma^n \beta \cdot \epsilon(\delta\epsilon)(\delta^2\epsilon) \cdot \ldots \cdot (\delta^{n-1}\epsilon) = \gamma^n \beta \prod_{k=0}^{n-1} \delta^k \epsilon.$$

Thus, the length of the trace is

$$|T_w| = |\gamma^n \beta| + \sum_{k=0}^{n-1} |\gamma^k \epsilon| = n + 1 + \sum_{k=0}^{n-1} (k+1) = \frac{1}{2}n^2 + \frac{3}{2}n + 1.$$

We can do a rapid test to see if we made a mistake: Is $(n^2 + 3n + 2)/2$ always an integer? This question boils down to: Is $n^2 + 3n$ even? That is: Do n^2 and $3n$ have the same parity? The answer is clearly yes, so $(n^2 + 3n + 2)/2$ is always an integer and we cannot deduce anything. (But it was worth a try.)

The delay $\mathcal{D}_n^{\texttt{isort}}$ laying obviously between the best and worst case delays, we deduce, for all $n \geqslant 0$,

$$2n + 1 \leqslant \mathcal{D}_n^{\texttt{isort}} \leqslant \tfrac{1}{2}n^2 + \tfrac{3}{2}n + 1.$$

This naturally raises the question as to whether the delay is more often closer to the lower bound or to the upper bound, because the former is linear whilst the latter is quadratic. (If the bounds were both linear or both quadratic, the issue would perhaps be a bit less critical when considering large values of n.) Let us assume that all integers have equal probability to be present at a given location in the input list and are pairwisely distinct. Let $\mathcal{A}_n^{\texttt{isort}}$ be the average delay of `isort(`L`)`, where the list L is comprised of n equiprobable distinct integers. Similarly, we write $\mathcal{A}_n^{\texttt{insert}}$. More specifically, the issue will hinge on the average number of times clause δ is used for an initial list of length n, therefore let us call $\mathcal{A}_n^{\delta}$ this important number. Let us recall here

```
isort(   [])                 -β-> [];
isort([I|L])                 -γ-> insert(I,isort(L)).
insert(I,[J|S]) when I > J   -δ-> [J|insert(I,S)];
insert(I,    S)              -ϵ-> [I|S].
```

We deduce the following equations from them:

$$\mathcal{A}_0^{\texttt{isort}} \overset{\beta}{=} 1; \quad \mathcal{A}_{k+1}^{\texttt{isort}} \overset{\gamma}{=} 1 + \mathcal{A}_k^{\texttt{insert}} + \mathcal{A}_k^{\texttt{isort}}, \text{ with } k \geqslant 0;$$
$$\mathcal{A}_0^{\texttt{insert}} = 1; \quad \mathcal{A}_k^{\texttt{insert}} = 1 + \mathcal{A}_k^{\delta}, \text{ from clauses } \delta \text{ and } \epsilon.$$

The term $\mathcal{A}_k^{\texttt{insert}}$ in equation $(\overset{\gamma}{=})$ is justified because the length of `isort(`L`)`, for all lists L, equals the length of L—here assumed to be k. Summing over k on both sides of $(\overset{\gamma}{=})$, we deduce

$$\sum_{k=0}^{n-1} \mathcal{A}_{k+1}^{\texttt{isort}} = \sum_{k=0}^{n-1} 1 + \sum_{k=0}^{n-1} \mathcal{A}_k^{\texttt{insert}} + \sum_{k=0}^{n-1} \mathcal{A}_k^{\texttt{isort}}.$$

By cancelling identical terms on both sides, we draw

$$\mathcal{A}_n^{\texttt{isort}} = n + \sum_{k=0}^{n-1} \mathcal{A}_k^{\texttt{insert}} + \mathcal{A}_0^{\texttt{isort}} = 1 + n + \sum_{k=0}^{n-1} \bigl(1 + \mathcal{A}_k^{\delta}\bigr) = 1 + 2n + \sum_{k=0}^{n-1} \mathcal{A}_k^{\delta}.$$

Let us determine now $\mathcal{A}_k^{\delta}$ by first asking ourselves how many ways are there to insert a number into a list of length k. Clearly, there are $k + 1$ possible locations: before the first item, before the second, etc., before the last *and after the last.* Secondly, assuming that the item is to be inserted at position j (the head of the list is at position 0), that is, it is aimed at being placed between the $(j-1)$th and jth element, how many times clause δ is needed? If inserted at location 0, clause δ is not used. If location 1, then it is used once etc. until location k, that is, after the last item, for which clause δ is utilised k times. Thirdly, each insertion position is equally likely because we posited the equal probability of occurrence for the integers in the list. Therefore, we can average the total number of times clause δ is used for all insertion positions by the number of such positions. In other words:

$$\mathcal{A}_k^{\delta} = \frac{1}{k+1}\sum_{j=0}^{k} j = \frac{k}{2}, \quad \mathcal{A}_k^{\texttt{insert}} = 1 + \mathcal{A}_k^{\delta} = \frac{k}{2} + 1. \tag{10.12}$$

This value could have been expected by means of the following informal reasoning. When considering the location to insert a number, since the numbers in the list are equiprobable, we should expect, in average, half of them to be lower than the number inserted. So, in average, the number of times clause δ is used is $k/2$. We can now substitute $\mathcal{A}_k^{\delta}$ by its simple closed form in the equation back and conclude

$$\mathcal{A}_n^{\texttt{isort}} = 1 + 2n + \sum_{k=0}^{n-1}\frac{k}{2} = \frac{1}{4}n^2 + \frac{7}{4}n + 1 \sim \frac{1}{4}n^2 \sim \frac{1}{2}\mathcal{W}_n^{\texttt{isort}}. \tag{10.13}$$

This comes as a disappointment, though, because it means that the average delay is quadratic, just as the worst case, even if the multiplicative constant is twice as small, asymptotically, that is, *the average delay $\mathcal{A}_n^{isort}$ is 50% of the worst delay $\mathcal{W}_n^{isort}$, for large values of n.*

We shall see later more asymptotically efficient sort algorithms, so it is worth wondering why insertion sort fails short to be a good algorithm, at least for large lists of integers. First, let us recall that items are inserted in reverse order because of clause γ. Thus, when clause δ applies, it means that `I` was originally before `J` and it is found now to be greater. After the rewrite, `J` is located before `I`. All this means that *one application of clause δ removes one inversion from the input.* In order to go faster, we need a sorting method that removes more than a linear number of inversions per rewrite. In the next section, we shall study such a faster method. As a corollary of this short analysis, the average number of inversions in a random permutation of n objects is

$$\sum_{k=0}^{n-1}\mathcal{A}_k^{\delta} = \sum_{k=0}^{n-1}\frac{k}{2} = \frac{n(n-1)}{4}.$$

Tail form. At this point, after so much mathematics, it is important to check our results. Another verification of the previously assessed delays can be done by modifying the definition of `isort/1` such that it ends with the sorted list *and* the number of performed rewrites. As we saw before, this addition is easily done if the original definition is already in tail form. Therefore, let us first transform `isort/1` into tail form. Starting from

`isort(   [])` $\xrightarrow{\beta}$ `[];`
`isort([I|L])` $\xrightarrow{\gamma}$ `insert(I,isort(L)).`
`insert(I,[J|S]) when I > J` $\xrightarrow{\delta}$ `[J|insert(I,S)];`
`insert(I,    S)` $\xrightarrow{\epsilon}$ `[I|S].`

we add a tuple accumulator to our functions and initialise it with the empty tuple (new clause α):

`isort_tf(L)` $\xrightarrow{\alpha}$ **`isort(L,{}).`**
`isort(   [],`**`A`**`)` $\xrightarrow{\beta}$ `[];` `% A` *`unused yet`*
`isort([I|L],`**`A`**`)` $\xrightarrow{\gamma}$ `insert(I,isort(L,`**`A`**`),`**`A`**`).`
`insert(I,[J|S],`**`A`**`) when I > J` $\xrightarrow{\delta}$ `[J|insert(I,S,`**`A`**`)];`
`insert(I,    S,`**`A`**`)` $\xrightarrow{\epsilon}$ `[I|S].` `% A` *`unused yet`*

We can now inspect each clause and, depending on its body shape (that is: expression in tail form, either with or without a call, or not in tail form), some transformation is done. First, the body of clause β is in tail form and does not contain any function call. Thus, we transform it by applying an auxiliary function `appk/2`:

`isort(   [],A)` $\xrightarrow{\beta}$ **`appk([],A);`**

Next is clause γ, which is not in tail form. The first call to be evaluated is `isort(L,A)`, whose control context is `insert(I,␣,A)`. Let us keep the call whilst saving into the accumulator `A` the variable `I` needed to rebuild the control context later, in a new clause of function `appk/2`. This variable needs a priori to be tagged by some unique atom, say `k1`:

`isort([I|L],A)` $\xrightarrow{\gamma}$ `isort(L,`**`{k1,I,A}`**`).`
`appk(V,{k1,I,A})` $\rightarrow$ **`insert(I,V,A).`**

The following clause is δ, which is not in tail form. The sole call to be evaluated is `insert(I,S,A)`, whose control context is `[J|␣]`. Let us associate `J` with a unique atom `k2`, then save both of them in the accumulator `A` and, dually, add a clause to `appk/2` to reconstruct the erased control context:

`insert(I,[J|S],A) when I > J` $\xrightarrow{\delta}$ `insert(I,S,`**`{k2,J,A}`**`);`
`appk(V,{k2,J,A})` $\rightarrow$ **`appk([J|V],A);`**

The last clause is ϵ, which is in tail form and contains no call, so we must apply its body to `appk/2` in order to check whether there are pending control contexts to rebuild:

```
insert(I,    S,A)             $\xrightarrow{\epsilon}$ appk([I|S],A).
```

In order to complete the transformation, we must add a clause to `appk/2` to process the case when the accumulator is empty, so the final result is found. Finally, the resulting program is (last step in bold)

```
isort_tf(L)                   $\xrightarrow{\alpha}$ isort(L,{}).
isort(   [],A)                $\xrightarrow{\beta}$ appk([],A);
isort([I|L],A)                $\xrightarrow{\gamma}$ isort(L,{k1,I,A}).
insert(I,[J|S],A) when I > J  $\xrightarrow{\delta}$ insert(I,S,{k2,J,A});
insert(I,    S,A)             $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,      {})              $\xrightarrow{\zeta}$ V;
appk(V,{k2,J,A})              $\xrightarrow{\eta}$ appk([J|V],A);
appk(V,{k1,I,A})              $\xrightarrow{\theta}$ insert(I,V,A).
```

We can remark that the atom `k1` is not necessary in the definition of `isort_tf/1`, since all other values in the accumulator are tagged `k2`:

```
isort_tf(L)                   $\xrightarrow{\alpha}$ isort(L,{}).
isort(   [],A)                $\xrightarrow{\beta}$ appk([],A);
isort([I|L],A)                $\xrightarrow{\gamma}$ isort(L,{I,A}).          % Here
insert(I,[J|S],A) when I > J  $\xrightarrow{\delta}$ insert(I,S,{k2,J,A});
insert(I,    S,A)             $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,      {})              $\xrightarrow{\zeta}$ V;
appk(V,{k2,J,A})              $\xrightarrow{\eta}$ appk([J|V],A);
appk(V,   {I,A})              $\xrightarrow{\theta}$ insert(I,V,A).       % and here
```

It becomes obvious now that `isort/2` reverses its first argument in the accumulator, which is initialised in clause α to the empty list. For instance, `isort([3,8,2],{})` $\xrightarrow{3}$ `appk([],{2,{8,{3,{}}}})`. Then it calls `appk/2` with the same arguments, in clause β. Hence, we would like to conclude that

`isort(`L`,[])` $\equiv$ `appk([],rev(`L`)),`

which would allow us to cut out the definition of `isort/2` entirely. The difficulty is that L is not a list, but a tuple. Of course, we could write a function whose special purpose would be to reverse this particular accumulator, but this lacks a great deal of generality. We must realise that, by opting for an accumulator implemented as a tuple, instead of a list, we optimised too early in terms of memory, which impedes now our effort to legitimately optimise the delay. Let us remember this lesson and revert to a list accumulator. When we are done with our delay optimisations, we shall revert to tuples. Let us resume from the definition

```
isort_tf(L)                  $\xrightarrow{\alpha}$ isort(L,[]).
isort(   [],A)               $\xrightarrow{\beta}$ appk([],A);
isort([I|L],A)               $\xrightarrow{\gamma}$ isort(L,[I|A]).
insert(I,[J|S],A) when I > J $\xrightarrow{\delta}$ insert(I,S,[{k2,J}|A]);
insert(I,    S,A)            $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,        [])           $\xrightarrow{\zeta}$ V;
appk(V,[{k2,J}|A])           $\xrightarrow{\eta}$ appk([J|V],A);
appk(V,     [I|A])           $\xrightarrow{\theta}$ insert(I,V,A).
```

Now we can proceed and remove `isort/2` as follows:

```
isort_tf(L)                  $\xrightarrow{\alpha}$ appk([],rev(L)).
insert(I,[J|S],A) when I > J $\xrightarrow{\delta}$ insert(I,S,[{k2,J}|A]);
insert(I,    S,A)            $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,      {})             $\xrightarrow{\zeta}$ V;
appk(V,{k2,J,A})             $\xrightarrow{\eta}$ appk([J|V],A);
appk(V,   {I,A})             $\xrightarrow{\theta}$ insert(I,V,A).
```

We expect that sorting a list or the same list reversed is the same:

$$\texttt{isort\_tf(}L\texttt{)} \equiv \texttt{isort\_tf(rev(}L\texttt{))}.$$

By clause α, this implies

$$\texttt{isort\_tf(}L\texttt{)} \equiv \texttt{appk([],rev(rev(}L\texttt{)))},$$

and since `rev(rev(`L`))` $\equiv$ L, we can replace the body of clause α simply by `appk([],`**`L`**`)`. Note how clause θ is still useful. We have now

```
isort_tf(L)                  $\xrightarrow{\alpha}$ appk([],L).
insert(I,[J|S],A) when I > J $\xrightarrow{\delta}$ insert(I,S,[{k2,J}|A]);
insert(I,    S,A)            $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,        [])           $\xrightarrow{\zeta}$ V;
appk(V,[{k2,J}|A])           $\xrightarrow{\eta}$ appk([J|V],A);
appk(V,     [I|A])           $\xrightarrow{\theta}$ insert(I,V,A).
```

We can get a shorter program at the expense of more comparisons. Remark that when clause η applies, `J` is lower than the head of `V`, which exists because this clause is only used to compute the bodies of clauses ϵ and η, where the first argument is not the empty list. Therefore, `appl([J|V],A)` $\equiv$ `insert(J,V,A)`, because clause ϵ would apply. Accordingly, let us change clause η:

```
isort_tf(L)                  $\xrightarrow{\alpha}$ appk([],L).
insert(I,[J|S],A) when I > J $\xrightarrow{\delta}$ insert(I,S,[{k2,J}|A]);
insert(I,    S,A)            $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,        [])           $\xrightarrow{\zeta}$ V;
appk(V,[{k2,J}|A])           $\xrightarrow{\eta}$ insert(J,V,A);
appk(V,     [I|A])           $\xrightarrow{\theta}$ insert(I,V,A).
```

We can see clearly now that `appk/2` calls `insert/3` in the same way in clauses η and θ, which means that it is useless to tag `J` with `k2` and we can get rid of clause θ:

```
isort_tf(L)                  $\xrightarrow{\alpha}$ appk([],L).
insert(I,[J|S],A) when I > J $\xrightarrow{\delta}$ insert(I,S,[J|A]);        % Here
insert(I,    S,A)            $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,   [])                $\xrightarrow{\zeta}$ V;
appk(V,[J|A])                $\xrightarrow{\eta}$ insert(J,V,A).       % and here
```

Perhaps it is clearer to get rid of `appk/2` by integrating its two operations in `isort_tf/1` and `insert/3`. Let us split clauses α and ϵ to manifest the cases where, respectively, `L` and `A` are empty:

```
isort_tf(   [])                      $\xrightarrow{\alpha_0}$ appk([],[]);
isort_tf([I|L])                      $\xrightarrow{\alpha_1}$ appk([],[I|L]).
insert(I,[J|S],    A) when I > J     $\xrightarrow{\delta}$ insert(I,S,[J|A]);
insert(I,    S,[J|A])                $\xrightarrow{\epsilon_0}$ appk([I|S],[J|A]);
insert(I,    S,   [])                $\xrightarrow{\epsilon_1}$ appk([I|S],[]).
appk(V,   [])                        $\xrightarrow{\zeta}$ V;
appk(V,[J|A])                        $\xrightarrow{\eta}$ insert(J,V,A).
```

We can now replace the bodies of clauses α_0 and ϵ_1 by their value, as given by clause ζ, and we can remove ζ:

```
isort_tf(   [])                      $\xrightarrow{\alpha_0}$ [];
isort_tf([I|L])                      $\xrightarrow{\alpha_1}$ appk([],[I|L]).
insert(I,[J|S],    A) when I > J     $\xrightarrow{\delta}$ insert(I,S,[J|A]);
insert(I,    S,[J|A])                $\xrightarrow{\epsilon_0}$ appk([I|S],[J|A]);
insert(I,    S,   [])                $\xrightarrow{\epsilon_1}$ [I|S].
appk(V,[J|A])                        $\xrightarrow{\eta}$ insert(J,V,A).
```

We saved one rewrite in case the input list is empty. Lastly, the bodies of clauses α_1 and ϵ_0 can be replaced by their value, as given by clause η, which can be, finally, erased. We rename the accumulator `A` as `L`.

```
isort_tf(   [])                      $\xrightarrow{\alpha_0}$ [];
isort_tf([I|L])                      $\xrightarrow{\alpha_1}$ insert(I,[],L).
insert(I,[J|S],    L) when I > J     $\xrightarrow{\delta}$ insert(I,S,[J|L]);
insert(I,    S,[J|L])                $\xrightarrow{\epsilon_0}$ insert(J,[I|S],L);
insert(I,    S,   [])                $\xrightarrow{\epsilon_1}$ [I|S].
```

It is important to remember that these last steps, relative to the removal of tag `k2` and so forth, make sense only because, in assessing the delay, we take into account only the number of function calls, not the number of comparisons, which is now greater for not using the control context `[J|␣]` in the original clause δ of `insert/3`. In other words, the items saved in the accumulator in the new clause δ have to be re-inserted in clause ϵ_0. Note also that it makes no sense to use tuples instead of a list to accumulate *single* items because the memory consumption would be exactly the same.

The same analysis used for assessing the delay of `isort/1` applies here as well, except that the items are inserted in their original order. So when the items are sorted increasingly, the delay is here *maximum* (that is, clause δ is used maximally) and when it is sorted non-increasingly, the delay is *minimum* (that is, clause δ is never used). *If items are not repeated, the best case of `isort/1` is the worst case of `isort_tf/1` and the worst case of `isort/1` is the best case of `isort_tf/1`.*

This is true because "nondecreasing" means the same as "increasing" when there is no repetition and "non-increasing" means "decreasing" under the same assumption. In order to find the lowest delay of the final version of `isort_tf/1`, it is helpful to get first a better understanding of the computational process by unfolding a simple example like sorting `[4,3,2,1]`, which is a list sorted in decreasing order:

$$
\begin{array}{ll}
\texttt{isort\_tf([4,3,2,1])} & \xrightarrow{\alpha_1} \texttt{insert(4,\ \ \ \ \ [],[3,2,1])}\\
& \xrightarrow{\epsilon_0} \texttt{insert(3,\ \ \ \ [4],\ \ [2,1])}\\
& \xrightarrow{\epsilon_0} \texttt{insert(2,\ \ [3,4],\ \ \ \ [1])}\\
& \xrightarrow{\epsilon_0} \texttt{insert(1,[2,3,4],\ \ \ \ \ [])}\\
& \xrightarrow{\epsilon_1} \texttt{[1,2,3,4]}.
\end{array}
$$

Let us note $\mathcal{B}_n^{\texttt{isort\_tf}}$ the best case of n items to be sorted. We have $\mathcal{B}_0^{\texttt{isort\_tf}} = 1$ by clause α_0. Let us assume next that $n > 0$. Then

- clause α_1 is used once;
- clause δ is not used, since we assume here that the items are already sorted non-increasingly;
- clause ϵ_0 is used once for each item in its third argument, which, by clause α_1, means all items except the first, that is $n-1$ times;
- clause ϵ_1 is used once.

In sum, the execution trace is $\alpha_1\epsilon_0^{n-1}\epsilon_1$. The total delay is then $\mathcal{B}_n^{\texttt{isort\_tf}} = 1 + (n-1) + 1 = n + 1$, if $n > 0$. Since we found that $\mathcal{B}_0^{\texttt{isort\_tf}} = 1 = 0 + 1$, we can extend the previous formula to $n = 0$. This result can be related directly to $\mathcal{W}_n^{\texttt{isort}} = (n^2 + 3n + 2)/2$, because the best case of `isort_tf/1` corresponds to the worst case of `isort/1` when integers are not repeated. We can further reckon that this minimum delay for `isort_tf/1` is also an absolute minimum for a sorting algorithm when the input is sorted non-increasingly, because it is simply the delay needed to reverse the input. If we follow the growth of the second and third arguments of `insert(4,[],[3,2,1])`, we recognise the behaviour of the second and first argument, respectively, of `rev_join/2`, for which $\mathcal{D}_n^{\texttt{rev\_join}} = n+1$ (see equation (3.6) on page 71). Given a list L sorted non-increasingly, we have

$$\texttt{isort(}L\texttt{)} \equiv \texttt{rev(}L\texttt{)}.$$

Assuming now the worst case, let us note $\mathcal{W}_n^{\texttt{isort\_tf}}$ the delay of `isort_-tf(`L`)` where the list L contains n items in increasing order. For the empty list, the execution trace is α_0. For singletons, for example, `[5]`, it is $\alpha_1\epsilon_1$. To understand the general case $n > 1$, we can try

```
isort_tf([1,2,3,4]) $\xrightarrow{\alpha_1}$ insert(1,     [],[2,3,4])
                    $\xrightarrow{\epsilon_0}$ insert(2,    [1],  [3,4])
                    $\xrightarrow{\delta}$ insert(2,     [],[1,3,4])
                    $\xrightarrow{\epsilon_0}$ insert(1,    [2],  [3,4])
                    $\xrightarrow{\epsilon_0}$ insert(3,  [1,2],    [4])
                    $\xrightarrow{\delta}$ insert(3,    [2],  [1,4])
                    $\xrightarrow{\delta}$ insert(3,     [],[2,1,4])
                    $\xrightarrow{\epsilon_0}$ insert(2,    [3],  [1,4])
                    $\xrightarrow{\epsilon_0}$ insert(1,  [2,3],    [4])
                    $\xrightarrow{\epsilon_0}$ insert(4,[1,2,3],     [])
                    $\xrightarrow{\delta}$ insert(4,  [2,3],    [1])
                    $\xrightarrow{\delta}$ insert(4,    [3],  [2,1])
                    $\xrightarrow{\delta}$ insert(4,     [],[3,2,1])
                    $\xrightarrow{\epsilon_0}$ insert(3,    [4],  [2,1])
                    $\xrightarrow{\epsilon_0}$ insert(2,  [3,4],    [1])
                    $\xrightarrow{\epsilon_0}$ insert(1,[2,3,4],     [])
                    $\xrightarrow{\epsilon_1}$ [1,2,3,4].
```

Notice the interplay of clauses δ and ϵ_0. A series of application of clause δ ends with the second argument to be the empty list. This is because the effect of clause δ is to save the contents of this argument by reversing it on top of the third argument. In other words, in the worst case, clause δ is equivalent to

```
insert(I,[J|S],L) when I > J →
                                   insert(I,[],rev_join(S,[J|L]));
```

A sequence of δ is followed by a series of ϵ_0 *of same length*, followed by another ϵ_0 or ϵ_1. The reason is that clause ϵ_0 restores on top of the second argument the items saved previously by clause δ. Then, if there are some items left in the last argument (to be sorted), one more application of clause ϵ_0 is required, otherwise the program ends with clause ϵ_1, that is, the execution trace when $n > 1$ is

$$\alpha_1 \prod_{p=0}^{n-2} \left(\delta^p \epsilon_0^{p+1}\right) \cdot \delta^{n-1} \epsilon_0^{n-1} \cdot \epsilon_1 = \alpha_1 \prod_{p=0}^{n-2} ((\delta\epsilon_0)^p \epsilon_0) \cdot (\delta\epsilon_0)^{n-1} \cdot \epsilon_1.$$

This observation is the key for finding the exact delay in the worst case as it hints at counting the rewrite steps of clause δ and of clause ϵ_0 *together*, as evinced in the right-hand side of the equality. We can now directly derive the worst delay:

$$\begin{aligned}
\mathcal{W}_n^{\texttt{isort\_tf}} &= \left|\alpha_1 \prod_{p=0}^{n-2} ((\delta\epsilon_0)^p \epsilon_0) \cdot (\delta\epsilon_0)^{n-1} \cdot \epsilon_1\right| \\
&= |\alpha_1| + \left|\prod_{p=0}^{n-2} ((\delta\epsilon_0)^p \epsilon_0)\right| + \left|(\delta\epsilon_0)^{n-1}\right| + |\epsilon_1| \\
&= 1 + \sum_{p=0}^{n-2} |(\delta\epsilon_0)^p \epsilon_0| + (n-1)|\delta\epsilon_0| + 1 \\
\mathcal{W}_n^{\texttt{isort\_tf}} &= 1 + \sum_{p=0}^{n-2} (2p+1) + 2(n-1) + 1 = n^2 + 1.
\end{aligned}$$

Since the worst case of `isort_tf/1` and `isort/1` are identical, we can compare their delays in this case, for $n \geqslant 0$,

$$\mathcal{W}_n^{\texttt{isort}} = (n^2 + 3n + 2)/2, \qquad \mathcal{W}_n^{\texttt{isort\_tf}} = n^2 + 1.$$

Let us relate now the best and worst cases for both `isort/1` and `isort_-tf/1`, we have, for $n > 3$,

$$\mathcal{B}_n^{\texttt{isort\_tf}} < \mathcal{B}_n^{\texttt{isort}} < \mathcal{W}_n^{\texttt{isort}} < \mathcal{W}_n^{\texttt{isort\_tf}}.$$

If we note $\mathcal{D}_n^{\texttt{isort}}$ the delay of `isort/1` on an input of length n, these inequalities are equivalent to say

$$\mathcal{B}_n^{\texttt{isort\_tf}} < \mathcal{D}_n^{\texttt{isort}} < \mathcal{W}_n^{\texttt{isort\_tf}}.$$

This is the best we can do because we only have the obvious inequalities

$$\mathcal{B}_n^{\texttt{isort\_tf}} \leqslant \mathcal{D}_n^{\texttt{isort\_tf}} \leqslant \mathcal{W}_n^{\texttt{isort\_tf}},$$

which do not allow us to compare $\mathcal{D}_n^{\texttt{isort}}$ and $\mathcal{D}_n^{\texttt{isort\_tf}}$. In order to obtain a stronger result, we need an average delay analysis so we can tell apart `isort_tf/1` from `isort/1`. Indeed, it might be that, for a given input list of length n, most configurations of the input lead to a delay for `isort_-tf/1` which is actually lower than for `isort/1`. Let us note $\mathcal{A}_n^{\texttt{isort\_tf}}$ the average number of rewrites needed to compute `isort_tf`(L), where the length of list L is n. Similarly, we note $\mathcal{A}_{p,q}^{\texttt{insert}}$ for the average delay of the call `insert(`I`,`P`,`Q`)`, where list P has length p and list Q has length q. The short story is this: because the numbers are random, the average number of times clause δ is used is $p/2$. Since clause ϵ_0 purpose, as observed before, is to put back the numbers previously moved by clause δ, we expect, in average, the same number, $p/2$, plus 1, because clause ϵ_0 also prepares the possible following use of clause δ. In other words, the difference with the longest execution trace defining $\mathcal{W}_n^{\texttt{isort}}$ is that the subsequences $\delta\epsilon_0$ are expected to be 50% shorter in average, so the execution trace is, in average,

$$\alpha_1 \prod_{p=0}^{n-2} \left((\delta\epsilon_0)^{p/2}\epsilon_0\right) \cdot (\delta\epsilon_0)^{(n-1)/2} \cdot \epsilon_1,$$

from which we deduce the average delay for $n > 1$:

$$\mathcal{A}_n^{\texttt{isort\_tf}} = 1 + \sum_{p=0}^{n-2}\left(2 \cdot \frac{p}{2} + 1\right) + \left(2 \cdot \frac{n-1}{2}\right) + 1 = \frac{1}{2}n^2 + \frac{1}{2}n + 1.$$

Elegantly, this formula extends to cope with $n = 0, 1$ and we can compare now $\mathcal{A}_n^{\texttt{isort\_tf}}$ to $\mathcal{A}_n^{\texttt{isort}}$, for $n \geqslant 0$:

$$\mathcal{A}_n^{\texttt{isort\_tf}} = \tfrac{1}{2}n^2 + \tfrac{1}{2}n + 1 \sim \tfrac{1}{2}n^2 \sim 2 \cdot \mathcal{A}_n^{\texttt{isort}}, \text{ as } n \to \infty.$$

In other words, `isort_tf/1`, in spite of being optimised, is nevertheless 50% slower than the original function, *in average for large values of n.*

Adding a counter. Anyway, it is always a good idea to test this kind of result also by means of a little experiment which consists simply in modifying the source code so the function returns its normal result paired with the number of function calls that have been needed to find it. Since `isort_tf/1` is, by definition, in tail form, this transformation is straightforward as it consists merely in adding a parameter (the call counter) to all function calls and incrementing it each time an arrow is crossed (note that we append to the function names the string `_c`):

```
isort_tf_c(   [])              -> {[],1};
isort_tf_c([I|L])              -> insert(I,[],L,1).
insert(I,[J|S],L,C) when I > J -> insert(I,S,[J|L],C+1);
insert(I,S,[J|L],C)            -> insert(J,[I|S],L,C+1);
insert(I,S,   [],C)            -> {[I|S],C+1}.
```

We can deduce in the same manner a version of `isort/1` which computes the delay as well by recalling the result of the automatic transformation of `isort/1` into tail form, *before any improvement*, as shown on page 214:

```
isort_tf(L)                 $\xrightarrow{\alpha}$ isort(L,{}).
```

```
isort(   [],A)                $\xrightarrow{\beta}$ appk([],A);
isort([I|L],A)                $\xrightarrow{\gamma}$ isort(L,{k1,I,A}).
insert(I,[J|S],A) when I > J  $\xrightarrow{\delta}$ insert(I,S,{k2,J,A});
insert(I,    S,A)             $\xrightarrow{\epsilon}$ appk([I|S],A).
appk(V,       [])             $\xrightarrow{\zeta}$ V;
appk(V,{k2,J,A})              $\xrightarrow{\eta}$ appk([J|V],A);
appk(V,{k1,I,A})              $\xrightarrow{\theta}$ insert(I,V,A).
```

Then we can add the counters and be cautious so as not to count the rewrites by `appk/2`, which have no equivalent in the original definition:

```
isort_c(L)                      $\xrightarrow{\alpha}$ isort_c(L,[],0).
isort_c(   [],A,C)              $\xrightarrow{\beta}$ appk([],A,C+1);
isort_c([I|L],A,C)              $\xrightarrow{\gamma}$ isort_c(L,{k1,I,A},C+1).
insert(I,[J|S],A,C) when I > J  $\xrightarrow{\delta}$ insert(I,S,{k2,J,A},C+1);
insert(I,    S,A,C)             $\xrightarrow{\epsilon}$ appk([I|S],A,C+1).
appk(V,       [],C)             $\xrightarrow{\zeta}$ {V,C};
appk(V,{k2,J,A},C)              $\xrightarrow{\eta}$ appk([J|V],A,C);
appk(V,{k1,I,A},C)              $\xrightarrow{\theta}$ insert(I,V,A,C).
```

Two-way insertion sort. Let us consider again `isort_tf/1`, as defined on page 216:

```
isort_tf(   [])                   $\xrightarrow{\alpha_0}$ [];
isort_tf([I|L])                   $\xrightarrow{\alpha_1}$ insert(I,[],L).
insert(I,[J|S],    L) when I > J  $\xrightarrow{\delta}$ insert(I,S,[J|L]);
insert(I,    S,[J|L])             $\xrightarrow{\epsilon_0}$ insert(J,[I|S],L);
insert(I,    S,   [])             $\xrightarrow{\epsilon_1}$ [I|S].
```

A source of inefficiency stems from clause δ moving items from the output list, that is, the second argument of `insert/3` in clause ϵ_0, to the input list, that is, the third argument. After the item to be inserted has found its place, these items will be *moved back* to the output list. One simple way to avoid this back and forth consists in keeping an auxiliary list onto which push and pop items while searching for the right location for insertion. This way, the input list will always strictly decrease. The technical difference is that we have two lists in which to insert. The first list holds the smaller items, sorted non-increasingly, and the second one contains greater items, sorted nondecreasingly. The current sorted list would therefore be obtained by reversing the first list onto the second one. Technically, the first list is called a *reversed prefix* of the current output and the second is a *suffix*. For example, `[5,3,1]` and `[6,8,9]`, making the current `[1,3,5,6,8,9]`. Conceptually, this couple of lists implements one list with constant-time access to two successive, inner items. Alternatively, it can be thought as one list with a moving finger. Consider

```
i2w(L)                                $\xrightarrow{\zeta}$ i2w([],[],L).
i2w(   [],    Q,     [])              $\xrightarrow{\alpha}$ Q;
i2w([I|P],    Q,     [])              $\xrightarrow{\beta}$ i2w(    P,[I|Q],[]);
i2w(    P,[J|Q],L=[K|_]) when J < K   $\xrightarrow{\gamma}$ i2w([J|P],    Q, L);
i2w([I|P],    Q,L=[K|_]) when K < I   $\xrightarrow{\delta}$ i2w(    P,[I|Q], L);
i2w(    P,    Q,  [K|R])              $\xrightarrow{\epsilon}$ i2w(    P,[K|Q], R).
```

Notice how this is a definition that comes out naturally in tail form because it is a refinement of a definition itself in tail form. Clauses γ and δ are in charge of finding the right position for insertion, whilst clause ϵ inserts the item. There are two possible places for that, because we have two lists and we chose to implement the insertion by pushing the item on top of the second list, that is, on the suffix. It would have been also correct to push it on the reversed prefix:

```
i2w(    P,    Q,  [K|R]) $\xrightarrow{\epsilon}$ i2w([K|P],   Q, R)
```

When the insertions are over, we must construct the final sorted list with the reversed prefix and the suffix. This situation is taken care of by clauses α and β, because their third argument is the empty list (no more items left to be inserted). We could call `rev_join/2`, as defined on page 60:

```
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

since it is exactly what we need, but, for didactic purposes, we prefer to keep the definition of `i2w/3` self-contained, whence clauses α and β. Note that, because clause ϵ grows the suffix of the result, we must reverse the first argument on top of the second to compute the result. Let us unroll a short example.

```
i2w([2,3,1,4]) $\xrightarrow{\zeta}$ i2w(     [],       [],[2,3,1,4])
               $\xrightarrow{\epsilon}$ i2w(     [],      [2],  [3,1,4])
               $\xrightarrow{\gamma}$ i2w(    [2],       [],  [3,1,4])
               $\xrightarrow{\epsilon}$ i2w(    [2],      [3],    [1,4])
               $\xrightarrow{\delta}$ i2w(     [],    [2,3],    [1,4])
               $\xrightarrow{\epsilon}$ i2w(     [],  [1,2,3],      [4])
               $\xrightarrow{\gamma}$ i2w(    [1],    [2,3],      [4])
               $\xrightarrow{\gamma}$ i2w(  [2,1],      [3],      [4])
               $\xrightarrow{\gamma}$ i2w([3,2,1],       [],      [4])
               $\xrightarrow{\epsilon}$ i2w([3,2,1],      [4],       [])
               $\xrightarrow{\beta}$ i2w(  [2,1],    [3,4],       [])
               $\xrightarrow{\beta}$ i2w(    [1],  [2,3,4],       [])
```

$\xrightarrow{\beta}$ `i2w(     [],[1,2,3,4],       [])`
$\xrightarrow{\alpha}$ `[1,2,3,4].`

In short, the execution trace is $(\zeta)(\epsilon)(\gamma\epsilon)(\delta\epsilon)(\gamma^3\epsilon)(\beta^3\alpha)$. Note how the number of times clause β is used is the number of items left on the left list. Clause α is used exactly once.

Let us find the delay in the best, worst and average delays for an input list of n items, that is, counting the function calls instead of the comparisons. The best case will exert minimally clauses γ and δ and this minimum number of calls turns out to be zero when the two comparisons are false. The first item inserted does not use clauses γ and δ, but only clause ϵ, so, right after, the reversed prefix is empty and the suffix contains this item. If we want to insert the second item without moving the contents of the lists, and go straight to use clause ϵ, the second item must be smaller than the first. Based on the same argument, the third item must be smaller than the second etc. In the end, this means that *the input, in the best case, is a list sorted non-increasingly.* This is the same best case as for `isort_tf/1`. The last steps consisting in the reversal of the prefix, such prefix being empty, we do not even use clause β at all—only clause α once. In other words, if we note $\mathcal{B}_n^{\texttt{i2w}}$ the delay when the input is a list of n items in non-increasing order, then, because the execution trace is $\zeta\epsilon^n\alpha$, we have

$$\mathcal{B}_n^{\texttt{i2w}} = n + 2.$$

Let us assume that the input list is noted `[`$a_0, a_1, \ldots, a_{n-1}$`]`. The worst case must exert clauses γ and δ, on the one hand, and clauses α and β, on the other hand, maximally. Let us focus first on maximising the use of γ and δ. Since a_0 is the first item, it is always pushed on the suffix list by clause ϵ. The second item, a_1, in order to travel the furthest, has to be inserted below a_0. By doing so, clause γ is used once and then ϵ, therefore, as a result, a_0 is on the left (the reversed prefix, which is the first argument) and a_1 on the right (the suffix, which is the second argument). In other words: we have `[`a_0`]` and `[`a_1`]`. Because of this symmetry, in pursuit of the worst case, we can now move either a_0 or a_1 to the facing list, that is, choose either to set $a_2 < a_0$ or $a_1 < a_2$.

- If $a_2 < a_0$, clause δ is used once, then clause ϵ. As a result, we have the configuration `[]` and `[`a_2`,`a_0`,`a_1`]`. This translates as $a_2 < a_0 < a_1$. The fourth item, a_3, must be inserted at the bottom of the right list, which must be first reversed on top of the left list by clause γ: we then have `[`a_1`,`a_0`,`a_2`]` and `[`a_3`]`. In other words: $a_2 < a_0 < a_1 < a_3$. Finally, the left list is reversed on top of the second by clause β

$$\begin{tikzcd}
a_3 \arrow[rrrrrrrr, bend left] & < & a_0 \arrow[rr, bend right] & < & a_1 \arrow[rr, bend left] & < & a_2 \arrow[llllll, bend left] & < & a_4
\end{tikzcd}$$

FIGURE 58: Worst case for `i2w/1` if $n = 5$ $(a_1 < a_2)$

and clause α is last. The execution trace is $(\zeta)(\epsilon)(\gamma\epsilon)(\delta\epsilon)(\gamma^3\epsilon)(\beta^3\alpha)$, whose length is 14.

- If $a_1 < a_2$, we would have `[`a_1`,`a_0`]` and `[`a_2`]`, then the lists `[]` and `[`a_3`,`a_0`,`a_1`,`a_2`]`, that is, $a_3 < a_0 < a_1 < a_2$. The complete execution trace: $(\zeta)(\epsilon)(\gamma\epsilon)(\gamma\epsilon)(\delta^2\epsilon)(\alpha)$. The length of this trace is 10, which is shorter than the previous trace.

As a conclusion, the choice $a_2 < a_0$ leads to a worse case. But what if the input list contains an odd number n of items? To guess what happens, let us insert a_4 assuming either $a_1 < a_2$ or $a_2 < a_0$.

- If $a_2 < a_0$, we should move out all the items of the left list, so we draw the configuration `[`a_4`]` and `[`a_2`,`a_0`,`a_1`,`a_3`]`: $a_4 < a_2 < a_0 < a_1 < a_3$, corresponding to the execution trace $(\zeta)(\epsilon)(\gamma\epsilon)(\delta\epsilon)(\gamma^3\epsilon)(\delta^3\epsilon)(\beta\alpha)$, whose length is 16.
- If $a_1 < a_2$, we want to insert a_4 at the bottom of the second list, thus obtaining `[`a_2`,`a_1`,`a_0`,`a_3`]` and `[`a_4`]`: $a_3 < a_0 < a_1 < a_2 < a_4$, corresponding to the execution trace $(\zeta)(\epsilon)(\gamma\epsilon)(\gamma\epsilon)(\delta^2\epsilon)(\gamma^4\epsilon)(\beta^4\alpha)$, whose length is 19. It is perhaps better visualised by means of oriented edges, revealing a spiral in FIGURE 58 on page 224.

Therefore, it seems that when the number of items is odd, having $a_1 < a_2$ leads to the worst delay, whilst $a_2 < a_0$ leads to the worst delay when the number of items is even. Let us determine these delays for any n and determine which is the greater. Let us note $\mathcal{W}_{2p+1}^{a_1<a_2}$ the former delay and $\mathcal{W}_{2p}^{a_2<a_0}$ the latter.

- If $n = 2p + 1$ and $a_1 < a_2$, then the execution trace is

  $$(\zeta)(\epsilon)(\gamma\epsilon)(\gamma\epsilon)(\delta^2\epsilon)(\gamma^4\epsilon)(\delta^4\epsilon)\ldots(\gamma^{2p-2}\epsilon)(\delta^{2p-2}\epsilon)(\gamma^{2p}\epsilon)(\beta^{2p}\alpha),$$

  as a run with $p = 3$ suggests:

  $$\begin{aligned}
  &\texttt{i2w([}a_0\texttt{,}a_1\texttt{,}a_2\texttt{,}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}\\
  &\xrightarrow{\zeta} \texttt{i2w([],[],[}a_0\texttt{,}a_1\texttt{,}a_2\texttt{,}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}\\
  &\xrightarrow{\epsilon} \texttt{i2w([],[}a_0\texttt{],[}a_1\texttt{,}a_2\texttt{,}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}\\
  &\xrightarrow{\gamma} \texttt{i2w([}a_0\texttt{],[],[}a_1\texttt{,}a_2\texttt{,}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}\\
  &\xrightarrow{\epsilon} \texttt{i2w([}a_0\texttt{],[}a_1\texttt{],[}a_2\texttt{,}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}\\
  &\xrightarrow{\gamma} \texttt{i2w([}a_1\texttt{,}a_0\texttt{],[],[}a_2\texttt{,}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}\\
  &\xrightarrow{\epsilon} \texttt{i2w([}a_1\texttt{,}a_0\texttt{],[}a_2\texttt{],[}a_3\texttt{,}a_4\texttt{,}a_5\texttt{,}a_6\texttt{])}
  \end{aligned}$$

```
$\xrightarrow{\delta^2}$ i2w([],[$a_0$,$a_1$,$a_2$],[$a_3$,$a_4$,$a_5$,$a_6$])
$\xrightarrow{\epsilon}$ i2w([],[$a_3$,$a_0$,$a_1$,$a_2$],[$a_4$,$a_5$,$a_6$])
$\xrightarrow{\gamma^4}$ i2w([$a_2$,$a_1$,$a_0$,$a_3$],[],[$a_4$,$a_5$,$a_6$])
$\xrightarrow{\epsilon}$ i2w([$a_2$,$a_1$,$a_0$,$a_3$],[$a_4$],[$a_5$,$a_6$])
$\xrightarrow{\delta^4}$ i2w([],[$a_3$,$a_0$,$a_1$,$a_2$,$a_4$],[$a_5$,$a_6$])
$\xrightarrow{\epsilon}$ i2w([],[$a_5$,$a_3$,$a_0$,$a_1$,$a_2$,$a_4$],[$a_6$])
$\xrightarrow{\gamma^6}$ i2w([$a_4$,$a_2$,$a_1$,$a_0$,$a_3$,$a_5$],[],[$a_6$])
$\xrightarrow{\epsilon}$ i2w([$a_4$,$a_2$,$a_1$,$a_0$,$a_3$,$a_5$],[$a_6$],[]).
```

If we omit clauses ζ, ϵ, α and β, we can see a pattern emerge from the sub-trace $(\gamma^2\delta^2)(\gamma^4\delta^4)(\gamma^6\delta^6)\dots(\gamma^{2p-2}\delta^{2p-2})(\gamma^{2p})$. Clause ϵ is used n times because it inserts the item in the right place. So the total delay is

$$\begin{aligned}\mathcal{W}_{2p+1}^{a_1<a_2} &= |\zeta| + |\epsilon^{2p+1}| + \sum_{k=1}^{p-1}\left(|\gamma^{2k}| + |\delta^{2k}|\right) + |\gamma^{2p}| + |\beta^{2p}\alpha|\\ &= 1 + (2p+1) + \sum_{k=1}^{p-1} 2(2k) + (2p) + (2p+1) = 2p^2 + 4p + 3.\end{aligned}$$

- If $n = 2p$, and $a_2 < a_0$, then the execution trace is

  $$(\zeta)(\epsilon)(\gamma\epsilon)(\delta\epsilon)(\gamma^3\epsilon)(\delta^3\epsilon)\dots(\gamma^{2p-1}\epsilon)(\beta^{2p-1}\alpha),$$

  as the following run with $p = 3$ suggests (first difference with the previous case is in bold):

```
i2w([$a_0$,$a_1$,$a_2$,$a_3$,$a_4$,$a_5$])
$\longrightarrow$ i2w(            [],               [],[$a_0$,$a_1$,$a_2$,$a_3$,$a_4$,$a_5$])
$\xrightarrow{\epsilon}$ i2w(            [],            [$a_0$], [$a_1$,$a_2$,$a_3$,$a_4$,$a_5$])
$\xrightarrow{\gamma}$ i2w(         [$a_0$],               [], [$a_1$,$a_2$,$a_3$,$a_4$,$a_5$])
$\xrightarrow{\epsilon}$ i2w(         [$a_0$],            [$a_1$],    [$a_2$,$a_3$,$a_4$,$a_5$])
$\xrightarrow{\delta}$ i2w(            [],       [$\boldsymbol{a_0}$,$\boldsymbol{a_1}$],   [$\boldsymbol{a_2}$,$\boldsymbol{a_3}$,$\boldsymbol{a_4}$,$\boldsymbol{a_5}$])
$\xrightarrow{\epsilon}$ i2w(            [],      [$a_2$,$a_0$,$a_1$],       [$a_3$,$a_4$,$a_5$])
$\xrightarrow{\gamma^3}$ i2w(   [$a_1$,$a_0$,$a_2$],               [],       [$a_3$,$a_4$,$a_5$])
$\xrightarrow{\epsilon}$ i2w(   [$a_1$,$a_0$,$a_2$],            [$a_3$],          [$a_4$,$a_5$])
$\xrightarrow{\delta^3}$ i2w(            [],   [$a_2$,$a_0$,$a_1$,$a_3$],          [$a_4$,$a_5$])
$\xrightarrow{\epsilon}$ i2w(            [],[$a_4$,$a_2$,$a_0$,$a_1$,$a_3$],             [$a_5$])
$\xrightarrow{\gamma^5}$ i2w([$a_3$,$a_1$,$a_0$,$a_2$,$a_4$],               [],             [$a_5$])
$\xrightarrow{\epsilon}$ i2w([$a_3$,$a_1$,$a_0$,$a_2$,$a_4$],            [$a_5$],               [])
```

If we omit clauses ζ, ϵ, α and β, we can see a pattern emerge from the sub-trace $(\gamma^1\delta^1)(\gamma^3\delta^3)(\gamma^5\delta^5)\dots(\gamma^{2p-3}\delta^{2p-3})(\gamma^{2p-1})$. Clause ϵ is used n times because it inserts the item in the right place. So the total delay is

$$\begin{aligned}\mathcal{W}_{2p}^{a_2<a_0} &= |\zeta| + |\epsilon^{2p}| + \sum_{k=1}^{p-1}\left(|\gamma^{2k-1}| + |\delta^{2k-1}|\right) + |\gamma^{2p-1}| + |\beta^{2p-1}\alpha|\\ &= 1 + (2p) + \sum_{k=1}^{p-1} 2(2k-1) + (2p-1) + ((2p-1)+1)\\ &= 2p^2 + 2p + 2.\end{aligned}$$

These formulas hold for all $p \geqslant 0$. We can now conclude this section about the worst case of `i2w/1`:

- If $n = 2p$, the worst case happens when the items satisfy the total order $a_{2p} < a_{2p-2} < \cdots < a_0 < a_1 < a_3 < \cdots < a_{2p-3} < a_{2p-1}$ and $\mathcal{W}_{2p}^{\texttt{i2w}} = 2p^2 + 2p + 2$, that is, $\mathcal{W}_n^{\texttt{i2w}} = \frac{1}{2}n^2 + n + 2$.
- If $n = 2p+1$, they satisfy $a_{2p-1} < a_{2p-3} < \cdots < a_3 < a_0 < a_1 < a_2 < \cdots < a_{2p-2} < a_{2p}$ and $\mathcal{W}_{2p+1}^{\texttt{i2w}} = 2p^2 + 4p + 3$, that is, $\mathcal{W}_n^{\texttt{i2w}} = \frac{1}{2}n^2 + n + \frac{3}{2}$.

These two delays are asymptotically equivalent to the worst delay of `isort/1`, which is quadratic:

$$\mathcal{W}_n^{\texttt{i2w}} \sim \frac{1}{2}n^2 \sim \mathcal{W}_n^{\texttt{isort}}, \text{ as } n \to \infty.$$

Despite our efforts, the two-way insertion sort does not beat the one-way insertion sort in the worst case. Could it be better in average, though?

Let us opt for a direct approach, based on an enumeration, which avoids recurrence equations. Let $\mathcal{A}_n^{\texttt{i2w}}$ be the average delay of `i2w(`L`)`, where L is a list of n distinct integers uniformly chosen at random. Let $\mathcal{A}_n^{\gamma\delta\epsilon}$ be the average number of times clauses γ, δ and ϵ are used, that is, it represents the average delay required to insert a random number into two random lists of n numbers in total. Let $\mathcal{A}_n^{\alpha\beta}$ the average number of times clauses α and β are used, that is, it denotes the average delay for reversing the final reversed prefix. This decomposition of delays comes from the observation that the evaluation first makes one step in order to set the initial values of the two lists. Then each item from the input list is inserted. Finally, the left list is reversed on top of the second.

$$\mathcal{A}_0^{\texttt{i2w}} := 2; \quad \mathcal{A}_n^{\texttt{i2w}} := 1 + \sum_{k=0}^{n-1}\mathcal{A}_k^{\gamma\delta\epsilon} + \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{A}_k^{\alpha\beta}, \quad \text{for } n > 0.$$

The summation bounds of $\mathcal{A}_k^{\alpha\beta}$ deserve some precise justification. Summing from 0 to $n-1$ means that we know that the result of a random insertion may lead to configurations where the left list contains a number of items ranging from 0 to $n-1$. Actually, it means more than that, as it supposes that each post-insertion configuration is unique. Examining clause ϵ reminds us that we always insert on the right list. Is this going to tip the scale in favour of the right list? Or does that mean that, since we could also choose to insert on the left list, the problem is perfectly symmetric and, therefore, it does not matter in the end?

$$\begin{tikzcd}
 & & & {[\,],[a]} \arrow[dll, "\gamma"'] \arrow[drr, "\epsilon"] & & & \\
 & {[a],[\,]} \arrow[d, "\epsilon"'] & & & & {[\,],[b,a]} \arrow[dl, "\gamma"'] \arrow[dr, "\epsilon"] & \\
 & {[a],[b]} \arrow[ddl, "\gamma"'] \arrow[dd, "\delta"'] \arrow[ddr, "\epsilon"] & & & {[b],[a]} \arrow[ddl, "\gamma"'] \arrow[ddr, "\epsilon"] & & {[\,],[c,b,a]} \\
 & & & & & & (c,b,a) \\
{[b,a],[\,]} \arrow[dd, "\epsilon"'] & {[\,],[a,b]} \arrow[dd, "\epsilon"'] & {[a],[c,b]} & {[a,b],[\,]} \arrow[dd, "\epsilon"'] & & {[b],[c,a]} & \\
 & & (a,c,b) & & & (b,c,a) & \\
{[b,a],[c]} & {[\,],[c,a,b]} & & {[a,b],[c]} & & & \\
(a,b,c) & (c,a,b) & & (b,a,c) & & &
\end{tikzcd}$$

FIGURE 59: Unbalanced two-way insertions of integers a, b and c

If intuition deserts us, let us turn to FIGURE 59 on page 227 where a small, albeit not too small, example is fully detailed. We see how all permutations of three objects are generated once. A longer example would give us more confidence in concluding that, after an insertion in a configuration whose two lists contain a total of k numbers, all configurations of $k+1$ numbers are present exactly once, except the one where the second list is empty, which is missing. This last point had to be expected because the insertion always takes place on the second list. Therefore the summation bounds of $\mathcal{A}_k^{\alpha\beta}$ must be 0 and $n-1$ (but not n because the second list cannot be empty by construction) and the sum must be averaged by n because the result of n insertions on an empty configuration leads to n unique configurations of n numbers. Let us resume our calculations by noting that $\mathcal{A}_k^{\alpha\beta} = k+1$ and $\mathcal{A}_0^{\gamma\delta\epsilon} = 1$:

$$\mathcal{A}_n^{\mathtt{i2w}} = 1 + \left(\mathcal{A}_0^{\gamma\delta\epsilon} + \sum_{k=1}^{n-1} \mathcal{A}_k^{\gamma\delta\epsilon}\right) + \frac{1}{n}\sum_{k=0}^{n-1}(k+1)$$

$$= 1 + 1 + \sum_{k=1}^{n-1} \mathcal{A}_k^{\gamma\delta\epsilon} + \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{2}n + \frac{5}{2} + \sum_{k=1}^{n-1} \mathcal{A}_k^{\gamma\delta\epsilon}.$$

Let us note $\mathcal{A}_{p,q}^{\gamma\delta\epsilon}$ the average delay for inserting one number in a configuration where the left list is made of p numbers and the right list contains q numbers, all of them being unique. Given the total number k of numbers in the two lists, all the possible values for p and q must satisfy $p + q = k$, with $p \geqslant 0$ and $q > 0$ (the second list can never be empty). So we have

$$\mathcal{A}_k^{\gamma\delta\epsilon} := \frac{1}{k}\sum_{p+q=k}^{q>0} \mathcal{A}_{p,q}^{\gamma\delta\epsilon} = \frac{1}{k}\sum_{q=1}^{k} \mathcal{A}_{k-q,q}^{\gamma\delta\epsilon}.$$

The reason for the division by k is that there are k possible configurations with a total of k numbers shared over the two lists. In order to compute $\mathcal{A}_{p,q}^{\gamma\delta\epsilon}$, given a configuration with p and q numbers, we have to enumerate all the places between two numbers where we can insert another one. Bear in mind that no number is actually slipped in, since insertion always occurs on the top of the second list by clause ϵ; what we really mean is that these places are possible locations for insertion in the total order corresponding to the configuration at hand.

For example, if the two lists are `[6,4,0]` and `[8,9]`, target locations for an insertion are evinced by a place-holder `␣` in the left list as `[6,␣,4,0]`, `[6,4,␣,0]` and `[6,4,0,␣]`; in the right list, we have the possibilities `[␣,8,9]`, `[8,␣,9]` and `[8,9,␣]`. There are $p + q + 1$ such locations in general: bottom of the second list up to its top, accounting for $q+1$, then bottom of the first list up to the place after the first item (if any), accounting for p. The location on top of the first list cannot be accounted for since it corresponds to the same position in the total order as the top of the second list; in the previous example, there can only be one location for `␣` in `0` $<$ `4` $<$ `6` $<$ `␣` $<$ `8` $<$ `9`, which consists in having `[6,4,0]` and `[␣,8,9]`, *not* `[␣,6,4,0]` and `[8,9]`.

The cumulated delay $\mathcal{A}_p^{\delta\epsilon}$ of targeting all the valid locations in a first list of length p is $\sum_{j=1}^{p}(j+1)$, where the index j ranges over the locations, 1 meaning "after the item at position 1," the summed j is the number of times clause δ is used and the summed 1 accounts for the use of clause ϵ. Similarly, the cumulated delay $\mathcal{A}_q^{\gamma\epsilon}$ of targeting all the valid locations in the second list of length q is $\sum_{j=0}^{q}(j+1)$, using j times clause γ instead of δ. Thus, we have

$$\mathcal{A}_{p,q}^{\gamma\delta\epsilon} := \frac{\mathcal{A}_p^{\delta\epsilon} + \mathcal{A}_q^{\gamma\epsilon}}{p+q+1} = \frac{1}{p+q+1}\left(\sum_{j=1}^{p}(j+1) + \sum_{j=0}^{q}(j+1)\right)$$

$$= \frac{1}{p+q+1}\left(\sum_{j=2}^{p+1} j + \sum_{j=1}^{q+1} j\right) = \frac{p^2+3p+q^2+3q+2}{2p+2q+2}.$$

Notice the complete symmetry of the expression: $\mathcal{A}^{\gamma\delta\epsilon}_{p,q} = \mathcal{A}^{\gamma\delta\epsilon}_{q,p}$. Anyway,

$$\mathcal{A}^{\gamma\delta\epsilon}_{k-q,q} = \frac{(k-q)^2+3(k-q)+q^2+3q+2}{2(k-q)+2q+2} = \frac{q^2}{k+1} - \frac{kq}{k+1} + \frac{k+2}{2}.$$

We can now replace $\mathcal{A}^{\gamma\delta\epsilon}_{k-q,q}$ in the definition of $\mathcal{A}^{\gamma\delta\epsilon}_{k}$:

$$\mathcal{A}^{\gamma\delta\epsilon}_{k} = \frac{1}{k}\left(\frac{1}{k+1}\sum_{q=1}^{k} q^2 - \frac{k}{k+1}\sum_{q=1}^{k} q + \frac{k+2}{2}\sum_{p=1}^{k} 1\right) = \frac{1}{3}k + \frac{7}{6}.$$

Finally, we can substitute $\mathcal{A}^{\gamma\delta\epsilon}_{k}$ for its newly found value in the definition of $\mathcal{A}^{\texttt{i2w}}_{n}$:

$$\mathcal{A}^{\texttt{i2w}}_{0} = 2; \quad \mathcal{A}^{\texttt{i2w}}_{n} = \frac{1}{2}n + \frac{5}{2} + \frac{1}{3}\sum_{k=1}^{n-1} k + \frac{7}{6}\sum_{k=1}^{n-1} 1 = \frac{1}{6}n^2 + \frac{3}{2}n + \frac{4}{3} \sim \frac{1}{6}n^2.$$

The bad news is that the average delay of `i2w/1` is quadratic, just as the other versions of insertion sort, but the good news is that the coefficient is smaller. Compare with

$$\mathcal{A}^{\texttt{isort}}_{n} = \tfrac{1}{4}n^2 + \tfrac{7}{4}n + 1 \text{ and } \mathcal{A}^{\texttt{isort\_tf}}_{n} = \tfrac{1}{2}n^2 + \tfrac{1}{2}n + 1.$$

Another remark is that the average delay becomes closer to 33% of the worst case as n grows, since we proved

$$\mathcal{W}^{\texttt{i2w}}_{n} \sim \tfrac{1}{2}n^2, \text{ as } n \to \infty.$$

Balanced two-way insertion sort. Is there a way to improve on the previous two-way insertion? One source of inefficiency stems from the fact that we wanted to retain the tail form, therefore an insertion in one of the two lists required, in general, to move some items to the other list. This results often in unbalanced lists, so subsequent insertions in the longest list will incur a greater delay. Let us design a *balanced two-way insertion*, where the lengths of the two lists differ by one at most and where we use the control context to hold the items during the one-way insertion phase in each of the two lists. To understand how this could work, let us imagine that we already have a configuration with two lists of same length. If we insert a number in the first list, the first list will jut over the second and, symmetrically, if we insert in the second, the second will stand higher than the first. In order to have some regularity and a shorter definition later, let us assume that, in this case, we want the second list to always be taller after the insertion. This enables dealing with only two situations instead of three, which are (1) the two lists have same length, (2) the second list is longer by one. Let us assume now that we insert a number when the second list is longer than the first one. Then, if the integer has to be inserted in the first list, the resulting lists will have equal lengths, which means we go back to case (1), otherwise, we move the top of the second list to the top of the first list in addition to the insertion itself, and we go back to case (1) as well. If we are in case (1) and the insertion takes place in the second stack, no rebalancing has to be done; otherwise, the top of the first stack is moved to the top of the second: in both events, we are in case (2). But what if the item to insert has to be simply pushed on top of one of the two lists? If the two lists have same length, let us push the number on top of the second one, otherwise it means that the second exceeds the first by one, so it is best to push it on the first list, so, as a result, the lists have equal lengths and we are back to case (1).

To implement this design, we shall need an additional parameter, which represents the difference in length between the lists. Since we opt for never having the first list jut over the second, this parameter will range just over two values exactly, more precisely, it will toggle between these two values: after an insertion, if the two lists had equal lengths, then the second one will be taller, otherwise, they will be of equal length. Usually, this type of parameter should be bound to two axioms, with explicit names telling the story, like `balanced` and `unbalanced`, but here it is simple and quite meaningful to give a numeric interpretation of this parameter as the length of the second list minus the length of the first, so this difference is an integer whose value is either `0` or `1`. Because the first list is the reversed prefix and the second is the suffix of the current sorted list, we need two one-way insertions: `ins_dn(`I`,`S`)` inserts the number I into the list S which is sorted non-increasingly; `ins_-up(`I`,`S`)` inserts I into S sorted non-decreasingly. These two functions are easily defined, as the latter is none other than `insert/2` on page 209 in disguise:

```
insert(I,[J|S]) when I > J -> [J|insert(I,S)];
insert(I,    S)            -> [I|S].

ins_up(I,[J|S]) when I > J $\xrightarrow{\iota}$ [J|ins_up(I,S)];
ins_up(I,    S)            $\xrightarrow{\kappa}$ [I|S].
```

and the former is derived by inverting the comparison in the guard:

```
ins_dn(I,[J|S]) when J > I $\xrightarrow{\lambda}$ [J|ins_dn(I,S)];
ins_dn(I,    S)            $\xrightarrow{\mu}$ [I|S].
```

Now we add the difference parameter, which is initialised with `0` because the two lists are initially empty:

```
i2wb(L) -> i2wb([],[],L,0).
```

Function `i2wb/3` on page 222 consists of two groups of clauses: clauses from γ to ϵ perform the comparisons, moving around the numbers until the insertion, properly speaking, takes place (a simple push on the right list); clauses α and β perform the final steps consisting in reversing the first list on top of the second and returning it as the result. The definition of `i2wb/4` will contain one more group of rules, stemming from the splitting of the group dedicated to comparisons into two subgroups, depending on the difference of lengths of the two lists, named `D` in the following code schema:

```
i2wb(   [],     Q,   [],_)              \xrightarrow{\alpha} Q;
i2wb([I|P],     Q,   [],D)              \xrightarrow{\beta} i2wb(P,[I|Q],[],D);

i2wb(    P,[J|Q],[K|R],0) when J < K \xrightarrow{\gamma} \boxed{\phantom{xxxxxxxxxxxx}};
i2wb([I|P],    Q,[K|R],0) when K < I \xrightarrow{\delta} \boxed{\phantom{xxxxxxxxxxxx}};
i2wb(    P,    Q,[K|R],0)            \xrightarrow{\epsilon} \boxed{\phantom{xxxxxxxxxxxx}};

i2wb(    P,[J|Q],[K|R],1) when J < K \xrightarrow{\zeta} \boxed{\phantom{xxxxxxxxxxxx}};
i2wb([I|P],    Q,[K|R],1) when K < I \xrightarrow{\eta} \boxed{\phantom{xxxxxxxxxxxx}};
i2wb(    P,    Q,[K|R],1)            \xrightarrow{\theta} \boxed{\phantom{xxxxxxxxxxxx}}.
```

Clauses γ, δ and ϵ are respectively dual of ζ, η and θ, although details due to the difference in length will lead to small differences in the bodies. Let us start with the patterns matching the case when both lists have the same length, that is, `D` is `0`. Let us consider clause γ. It matches the case when the item to insert, `K`, belongs to the second list `[J|Q]`, so `ins_up/2` must be called. Since the second list will exceed the first by one, the recursive call must be done with a difference of `1`. Notice how `J` is already at the right place and does no need to be included in the call to `ins_up/2`.

```
i2wb(P,[J|Q],[K|R],0) when J < K \xrightarrow{\gamma}
                                        i2wb(P,[J|ins_up(K,Q)],R,1);
```

At this point it may appear more obviously that our new design is based both on one-way insertion and unbalanced, two-way insertion. Clause δ processes the situation in which `K` has to be inserted in the first list. Because we decided to always have the second list of greater or equal length than the first, we need, in addition to a one-way insertion into a non-increasing list, to move the top of the first list, `I`, to the second list. The recursive call is subsequentially performed with a difference of length equal to `1`.

```
i2wb([I|P],Q,[K|R],0) when K < I $\xrightarrow{\delta}$
                                     i2wb(ins_dn(K,P),[I|Q],R,1);
```

Clause ϵ handles the case when `K` lays between the two lists. Following the same design principle to favour the second list in case of equal lengths, we push it on `Q` and call recursively with a difference of `1`.

```
i2wb(P,Q,[K|R],0) $\xrightarrow{\epsilon}$ i2wb(P,[K|Q],R,1);
```

Let us turn to the dual clauses where the difference of lengths is `1` instead of `0`. The difference between clauses γ and ζ is that the latter must reinstate the balance by moving the top of the second list, `J` to the top of the first, `P`. The recursive call is performed, consistently, with a difference of `0`, that is, the lists are balanced after insertion.

```
i2wb(P,[J|Q],[K|R],1) when J < K $\xrightarrow{\zeta}$
                                     i2wb([J|P],ins_up(K,Q),R,0);
```

The difference between clauses δ and η is that we do not move the topmost item of the first list because the second list is already taller. Therefore, we insert `K` in it, and that's it. The recursive call indicates that the lists are now balanced.

```
i2wb([I|P],Q,[K|R],1) when K < I $\xrightarrow{\eta}$
                                     i2wb([I|ins_dn(K,P)],Q,R,0);
```

Finally, the difference between clauses ϵ and θ is that we insert `K` on the first list because it is shorter by one, achieving thus the balance.

```
i2wb(P,Q,[K|R],1) $\xrightarrow{\theta}$ i2wb([K|P],Q,R,0).
```

As a summary, here is the full definition of `i2wb/4`:

```
i2wb(   [],    Q,   [],_)              $\xrightarrow{\alpha}$ Q;
i2wb([I|P],    Q,   [],D)              $\xrightarrow{\beta}$ i2wb(P,[I|Q],[],D);
i2wb(    P,[J|Q],[K|R],0) when J < K $\xrightarrow{\gamma}$
                                     i2wb(P,[J|ins_up(K,Q)],R,1);
i2wb([I|P],    Q,[K|R],0) when K < I $\xrightarrow{\delta}$
                                     i2wb(ins_dn(K,P),[I|Q],R,1);
i2wb(    P,    Q,[K|R],0)              $\xrightarrow{\epsilon}$ i2wb(P,[K|Q],R,1);
i2wb(    P,[J|Q],[K|R],1) when J < K $\xrightarrow{\zeta}$
                                     i2wb([J|P],ins_up(K,Q),R,0);
i2wb([I|P],    Q,[K|R],1) when K < I $\xrightarrow{\eta}$
                                     i2wb([I|ins_dn(K,P)],Q,R,0);
i2wb(    P,    Q,[K|R],1)              $\xrightarrow{\theta}$ i2wb([K|P],Q,R,0).
```

Let us consider in FIGURES 60 to 61 on page 233 all the different configurations for the two lists after the insertion of b, c and d. Let us

$$\begin{tikzcd}
 & & {[],[a]} \arrow[dl, "\zeta\kappa"'] \arrow[drr, "\theta"] & & & \\
 & {[a],[b]} \arrow[dl, "\delta\mu"'] \arrow[d, "\epsilon"] \arrow[dr, "\gamma\kappa"] & & & {[b],[a]} \arrow[dl, "\delta\mu"'] \arrow[d, "\epsilon"] \arrow[dr, "\gamma\kappa"] & \\
{[c],[a,b]} & {[a],[c,b]} & {[a],[b,c]} & {[c],[b,a]} & {[b],[c,a]} & {[b],[a,c]} \\
(c,a,b) & (a,c,b) & (a,b,c) & (c,b,a) & (b,c,a) & (b,a,c)
\end{tikzcd}$$

FIGURE 60: Balanced two-way insertions of b and c in `[],[`a`]`

wonder now what are the best, worst and average delays of `i2wb/1`. Let us assume that we have the input `[`a_0, a_1, a_2, a_3, a_4`]` and we want it to minimise the rewrites, which means not to use clauses γ, δ, ζ and η; also, the usage of clause β should be minimum. The latter clause is not an issue because it reverses the first list and, by design, the second list has the same length as the first, or exceeds it at most by one number. A simple diagram with the two lists initially empty suffices to convince us that the numbers must go alternatively to the right and then to the left, leading, for example, to `[`a_3, a_1`]` and `[`a_4, a_2, a_0`]`. This is perhaps better visualised by means of oriented edges revealing a whirlpool in FIGURE 62 on page 234, to be contrasted with the spiral in FIGURE 58 on page 224 for `i2w/1`.

The clause definining `i2w/1` has to be used first. Then each number is inserted, alternatively by means of clause ϵ and θ. Finally, the first list is reversed by clauses α and β, so the question hinges on determining the length of this list in the best case. By design, if the total number of items is even, then the two lists will end up containing, before using clause β, exactly half of them, because the lists have the same length. If the total is odd, the first list contains the quotient of this number when halved. Technically, let us note $\mathcal{B}_n^{\texttt{i2wb}}$ the delay of any call `i2wb(`L`)`, where the list L contains n numbers.

$$\mathcal{B}_{2p}^{\texttt{i2wb}} := 1 + 2p + p = 3p + 1,$$

$$\begin{tikzcd}
 & {[a],[b,c]} \arrow[dl, "\eta\mu"'] \arrow[d, "\theta"'] \arrow[dr, "\zeta\kappa"] \arrow[drr, "\zeta\iota\kappa"] & & \\
{[a,d],[b,c]} & {[d,a],[b,c]} & {[b,a],[d,c]} & {[b,a],[c,d]} \\
(d,a,b,c) & (a,d,b,c) & (a,b,d,c) & (a,b,c,d)
\end{tikzcd}$$

FIGURE 61: Balanced two-way insertion of d in `[`a`],[`b`,`c`]`

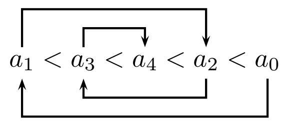


FIGURE 62: Best case for `i2wb/1` if $n = 5$

$$\mathcal{B}^{\texttt{i2wb}}_{2p+1} := 1 + (2p+1) + p = 3p + 2, \text{ with } p \geqslant 0.$$

Another, more compact, way to put it is: for $n \geqslant 0$,

$$\mathcal{B}^{\texttt{i2wb}}_{n} = 1 + n + \lfloor n/2 \rfloor \sim \tfrac{3}{2}n, \text{ as } n \to \infty.$$

where $\lfloor x \rfloor$, pronounced "floor x," is the greatest integer smaller than x. The equivalence is correct because $n/2 - 1 < \lfloor n/2 \rfloor \leqslant n/2$.

The worst case occurs when insertions are repeatedly performed at the bottom of the longest list. This case is left to the reader as an exercise.

Let us focus on the average delay. Let us suppose first the easiest case in which n is even. It is the easiest because, by design, both lists will have the same length after the insertions are done. Technically, this means that there exists an integer p such that $n = 2p$. Then we have

$$\mathcal{A}^{\texttt{i2wb}}_{0} := 2; \quad \mathcal{A}^{\texttt{i2wb}}_{2p} := 1 + \sum_{k=0}^{2p-1} \mathcal{A}_k + \mathcal{A}^{\alpha\beta}_{p}, \text{ for } p \geqslant 0,$$

where $\mathcal{A}_k$ is the average number of rewrites to insert a random number into a random configuration of two lists whose total number of items is k and $\mathcal{A}^{\alpha\beta}_{p}$ is the number of rewrites to reverse p numbers from the first list to the second and finally return the second list. The term $\mathcal{A}^{\alpha\beta}_{p}$ comes from the fact that, because $n = 2p$, there will be p numbers left after all the insertions are over. Obviously, we have

$$\mathcal{A}^{\alpha\beta}_{p} := p + 1.$$

The determination of $\mathcal{A}_k$ requires the consideration of only two cases: either k is even or it is odd. When analysing the average delay of `i2w/1`, there were much more configurations to take into account because not all the insertions lead to balanced lists. If k is even, then there exists an integer j such that $k = 2j$ and

$$\mathcal{A}_{2j} = \mathcal{A}_{j,j},$$

where $\mathcal{A}_{j,j}$ is the average number of rewrites to insert a random number into a configuration of two lists of length j. We already computed $\mathcal{A}_{p,q}$ on page 228 where we called it $\mathcal{A}^{\gamma\delta\epsilon}_{p,q}$ (we omit here the names of the clauses because there are too many involved):

$$\mathcal{A}_{p,q} = \frac{p^2 + 3p + q^2 + 3q + 2}{2p + 2q + 2}.$$

So we draw, by substitution and simplification,

$$\mathcal{A}_{2j} = \frac{j^2 + 3j + 1}{2j + 1} = \frac{1}{2}j - \frac{1}{4} \cdot \frac{1}{2j + 1} + \frac{5}{4}.$$

The remaining case is k being odd, that is, $k = 2j + 1$. By design, we know that the first list will hold j numbers whilst the second $j + 1$:

$$\mathcal{A}_{2j+1} = \mathcal{A}_{j,j+1} = \tfrac{1}{2}j + \tfrac{3}{2}.$$

Let us come back to our initial goal:

$$\mathcal{A}_{2p}^{\texttt{i2wb}} = 1 + \sum_{k=0}^{2p-1} \mathcal{A}_k + (p + 1).$$

In order to use $\mathcal{A}_{2j}$ and $\mathcal{A}_{2j+1}$, we must group the terms $\mathcal{A}_k$ pairwise, so we explicitly see that we sum successive even and odd numbers ($k = 2j$ and $k = 2j + 1$):

$$\mathcal{A}_{2p}^{\texttt{i2wb}} = 2 + p + \sum_{j=0}^{p-1} (\mathcal{A}_{2j} + \mathcal{A}_{2j+1}) = 2 + p + \sum_{j=0}^{p-1} \mathcal{A}_{2j} + \sum_{j=0}^{p-1} \mathcal{A}_{2j+1}.$$

Now we can make use of the values we found above for $\mathcal{A}_{2j}$ and $\mathcal{A}_{2j+1}$:

$$\begin{aligned}
\mathcal{A}_{2p}^{\texttt{i2wb}} &= 2 + p + \sum_{j=0}^{p-1} \left(\frac{1}{2}j - \frac{1}{4} \cdot \frac{1}{2j+1} + \frac{5}{4}\right) + \sum_{j=0}^{p-1} \left(\frac{1}{2}j + \frac{3}{2}\right) \\
&= 2 + p + \left(\frac{1}{2}\sum_{j=0}^{p-1} j - \frac{1}{4}\sum_{j=0}^{p-1} \frac{1}{2j+1} + \frac{5}{4}p\right) + \left(\frac{1}{2}\sum_{j=0}^{p-1} j + \frac{3}{2}p\right) \\
&= \frac{1}{2}p^2 + \frac{13}{4}p + 2 - \frac{1}{4}\sum_{j=0}^{p-1} \frac{1}{2j+1}.
\end{aligned}$$

We need to find the value of this sum. By definition, the nth *harmonic number*, noted H_n for all $n > 0$, is

$$H_n := \sum_{j=1}^{n} \frac{1}{j}. \text{ So, obviously, } H_{2p} = \sum_{j=1}^{2p} \frac{1}{j}.$$

Let us group in H_{2p}, on the one hand, the inverses with an odd numerator and, on the other hand, the inverses with an even numerator:

$$H_{2p} = \sum_{j=0}^{p-1} \frac{1}{2j+1} + \sum_{j=1}^{p} \frac{1}{2j} = \sum_{j=0}^{p-1} \frac{1}{2j+1} + \frac{1}{2}H_p.$$

We can express now our sum in terms of the harmonic series:

$$\sum_{j=0}^{p-1} \frac{1}{2j+1} = H_{2p} - \frac{1}{2}H_p.$$

We can now replace it in the latest equation defining $\mathcal{A}_{2p}^{\texttt{i2wb}}$:

$$\begin{aligned}\mathcal{A}_{2p}^{\texttt{i2wb}} &= \frac{1}{2}p^2 + \frac{13}{4}p + 2 - \frac{1}{4}\Big(H_{2p} - \frac{1}{2}H_p\Big)\\ &= \frac{1}{2}p^2 + \frac{13}{4}p + \frac{1}{8}H_p - \frac{1}{4}H_{2p} + 2.\end{aligned}$$

We can check this formula for $p = 0$, looking forward to the expected result 2. For the formula to work, we must extend H_p so that $H_0 = 0$. Then we indeed deduce $\mathcal{A}_0^{\texttt{i2wb}} = 2$. The remaining task consists in finding $\mathcal{A}_{2p+1}^{\texttt{i2wb}}$. Resuming the same line of thoughts, we deduce

$$\mathcal{A}_{2p+1}^{\texttt{i2wb}} = 1 + \sum_{k=0}^{2p} \mathcal{A}_k + \mathcal{A}_p^{\alpha\beta}, \text{ for } p \geqslant 0,$$

because we know that there will be p numbers left in the first list after the insertions are over. Instead of doing calculations almost similar to the previous ones, it is wise to reuse them:

$$\begin{aligned}\mathcal{A}_{2p+1}^{\texttt{i2wb}} &= 1 + \left(\sum_{k=0}^{2p-1} \mathcal{A}_k + \mathcal{A}_{2p}\right) + \mathcal{A}_p^{\alpha\beta} = \mathcal{A}_{2p}^{\texttt{i2wb}} + \mathcal{A}_{2p}\\ &= \left(\frac{1}{2}p^2 + \frac{13}{4}p + \frac{1}{8}H_p - \frac{1}{4}H_{2p} + 2\right) + \left(\frac{1}{2}p - \frac{1}{4}\cdot\frac{1}{2p+1} + \frac{5}{4}\right)\\ &= \frac{1}{2}p^2 + \frac{15}{4}p + \frac{1}{8}H_p - \frac{1}{4}H_{2p+1} + \frac{13}{4}.\end{aligned}$$

Let us try $p = 0$: $\mathcal{A}_1^{\texttt{i2wb}} = -\frac{1}{4}H_1 + \frac{13}{4} = 3$, which is correct. Summarising, for all $p \geqslant 0$:

$$\mathcal{A}_{2p}^{\texttt{i2wb}} = \frac{1}{2}p^2 + \frac{13}{4}p + \frac{1}{8}H_p - \frac{1}{4}H_{2p} + 2, \tag{10.14}$$

$$\mathcal{A}_{2p+1}^{\texttt{i2wb}} = \frac{1}{2}p^2 + \frac{15}{4}p + \frac{1}{8}H_p - \frac{1}{4}H_{2p+1} + \frac{13}{4}. \tag{10.15}$$

Let us determine the asymptotic behaviour of H_n. Perhaps the easiest way consists in proving that, for all nonzero real numbers x,

$$1 + x < e^x,$$

where e is an irrational constant whose approximate value is

$$e = 2.718281828459045\ldots$$

This is easily done by defining a real function ϕ such that $\phi(x) = e^x - x - 1$, and demonstrating that its values increase from a positive value. The derivative is $\phi'(x) = e^x - 1$, so $\phi'(x) > 0$ for all $x > 0$. Since $\phi(0) = 0$, the inequality above follows. The next step makes use of it to find a lower bound for H_n. As a special case, for all integers $i > 0$,

$$1 + \frac{1}{i} < e^{1/i}.$$

Since both sides of the inequality are positive, we can multiply all its instances for i ranging from 1 to n:

$$\prod_{i=1}^{n}\left(1 + \frac{1}{i}\right) < \prod_{i=1}^{n} e^{1/i}.$$

Equivalently and using the notation $\exp(x) := e^x$:

$$n + 1 = \frac{(n+1)!}{n!} = \prod_{i=1}^{n}\frac{i+1}{i} < \exp\left(\sum_{i=1}^{n}\frac{1}{i}\right) = \exp(H_n).$$

Let ln be the inverse function of exp. The number $\ln p$ is called the *natural logarithm* (or *Napierian logarithm*) of $p > 0$. It is an increasing function, so we can apply it to both sides of the inequation above and

$$\ln(n+1) < H_n.$$

An upper bound of H_n can be similarly obtained by replacing x by $-x$ in the original inequation $1 + x \leqslant e^x$, since it is valid for all $x \neq 0$, which yields $1 - x \leqslant e^{-x}$. Following now the same track as before, we set $x = 1/i$, for $i > 1$, and find, for $n > 1$, the equivalent inequalities:

$$1 - \frac{1}{i} < e^{-1/i},$$
$$\frac{1}{n} = \frac{(n-1)!}{n!} = \prod_{i=2}^{n}\frac{i-1}{i} < \exp\left(-\sum_{i=2}^{n}\frac{1}{i}\right) = \frac{1}{\exp(H_n - 1)},$$
$$\exp(H_n - 1) < n,$$
$$H_n < 1 + \ln n.$$

Therefore, we established the following inequalities for $n > 1$:

$$\ln(n+1) < H_n < 1 + \ln n.$$

We can now proceed with bounding $\mathcal{A}^{\mathsf{i2wb}}_{2p}$ and $\mathcal{A}^{\mathsf{i2wb}}_{2p+1}$:

$$\begin{aligned}\ln(p+1) - 2\ln(2p) &< 8 \cdot \mathcal{A}^{\mathsf{i2wb}}_{2p} - 4p^2 - 26p - 14\\ &< \ln p - 2\ln(2p+1) + 3;\\ \ln(p+1) - 2\ln(2p+1) &< 8 \cdot \mathcal{A}^{\mathsf{i2wb}}_{2p+1} - 4p^2 - 30p - 24\\ &< \ln p - 2\ln(2p+2) + 3.\end{aligned}$$

Setting $n = 2p$ and $n = 2p + 1$ leads, respectively, to the following bounds:

$$\begin{aligned}-2\ln n + \ln(n+2) + 4 < \varphi(n) < -2\ln(n+1) + \ln n + 7,\\ -2\ln n + \ln(n+1) < \varphi(n) < -2\ln(n+1) + \ln(n-1) + 3,\end{aligned}$$

where $\varphi(n) := 8 \cdot \mathcal{A}^{\mathsf{i2wb}}_{n} - n^2 - 13n - 10 + \ln 2$. We retain the minimum of the lower bounds and the maximum of the upper bounds of $\varphi(n)$ so, for all $n > 0$,

$$\ln(n+1) - 2\ln n < \varphi(n) < -2\ln(n+1) + \ln n + 7.$$

We can weaken the bounds a little bit with $\ln n < \ln(n+1)$ and simplify into

$$0 < 8 \cdot \mathcal{A}^{\mathsf{i2wb}}_{n} - n^2 - 13n + \ln n - 10 + \ln 2 < 7.$$

In other words, for any $n \in \mathbb{N}$, there exists ϵ_n such that $0 < \epsilon_n < 7/8$ and

$$\mathcal{A}^{\mathsf{i2wb}}_{n} = \tfrac{1}{8}(n^2 + 13n - \ln n + 10 - \ln 2) + \epsilon_n. \qquad (10.16)$$

Now we need bounds on $\ln n$. Let us start by proving that, for any $x \geqslant 0, x^2 < e^x$. Let $\psi(x) = e^x - x^2$. We have $\psi'(x) = e^x - 2x > 0$ for all x (see above how we dealt with $1 + x < e^x$) and $\psi(0) = 1$, thus $\psi(x) > 0$ for $x \geqslant 0$. Next, we can replace x by $\sqrt{y}$, when $x > 1$, yielding the inequalities

$$\begin{aligned}1 < y < e^{\sqrt{y}} \Rightarrow\ & 0 < \ln y < \sqrt{y} \ \Rightarrow\ 0 < (\ln y)/y < 1/\sqrt{y}\\ \Rightarrow\ & (\ln y)/y \sim 0. \qquad (10.17)\end{aligned}$$

We conclude

$$\mathcal{A}^{\mathsf{i2wb}}_{n} \sim \tfrac{1}{8}n^2, \text{ as } n \rightarrow \infty.$$

This is the best, that is, lowest, asymptotic average delay we have found so far and we conjecture that this is the best we can hope for with two-way insertion sort. Of course, the bad news is that we could not beat the quadratic delay in itself and, therefore, we shall study more efficient sorting algorithms in a later section.

Proving correctness. Let us recall the original version of `isort/1`:

```
-module(isort).
-export([isort/1]).

isort(   []) -> [];
isort([I|L]) -> insert(I,isort(L)).

insert(I,[J|S]) when I > J -> [J|insert(I,S)];
insert(I,    T)            -> [I|T].
```

(We renamed `S` into `T` in the second clause of `insert/2` to make the following proof easier to understand.) On page 40 we introduced informally the concept of correctness as being the property of some function to compute what it is expected to compute and not to compute anything it is not expected to. Therefore, we should define precisely what it means for `isort/1` to be correct. (In the following, we shall assume that "sorted list" means "nondecreasingly sorted list." The same proof can be done for non-increasingly sorted lists.) The function at hand is supposed to implement a sorting algorithm, so we could say that we want to make sure that the result is a sorted list and that it contains exactly the same items as the input. Let us make this statement more precise and reckon by induction on the length of the list.

First, let us check that `isort([])` is a sorted list: the first clause tells us that the result is `[]`, as expected.

Now, let us posit the *induction hypothesis* that `isort/1` is correct for all lists of a given length $n > 0$ and let us endeavour to prove the same for all lists of length $n + 1$. If we succeed, the *induction principle* would allow us to conclude that `isort/1` is correct for all inputs.

Let us suppose that we have an arbitrary list L of length $n > 0$ and an arbitrary item I, comparable to all items in L. Then, the second clause defining `isort/1` implies `insert(`I`,isort(`L`))`. The induction hypothesis applies to the call `isort(`L`)`, so it is correct; hence we only need to establish that the call to `insert/2` is correct. But we have not defined yet what it precisely means for `insert/2` to be correct. Let us say that it signifies that, given a sorted list T and an item I comparable to all items in T, the call `insert(`I`,`T`)` results in a sorted list containing I and all the items of T.

We have to do a proof by *complete induction* on this function, based on the length of T. The second clause defining `insert/2` entails that `insert(`I`,[])` evaluates in the expected result `[`I`]`: this is the base case. The complete induction hypothesis for `insert/2` is that `insert(`I`,`T`)` is correct for all lists T whose length is lower or equal than a given $n > 0$

and for all items I comparable with all the items in T.

Let us prove that `insert(`I`,`U`)` is correct, where the arbitrary sorted list U has length $n+1$. If we succeed, the induction principle would imply that `insert/1` is correct for all inputs. Let us suppose given an arbitrary list T of length $n > 0$ and a totally comparable item I. Because T is not empty, it has a first item J and an immediate sub-list S, that is, T = `[`J`|`S`]`. Item I is comparable to J, so two cases are possible: either $I \leqslant J$ or $I > J$. If the former, the second clause applies and ends with `[`I`|`T`]`. This list has length $n+1$, it contains all the items it should, it is sorted because $I < J$ and T being sorted entail that I is lower than all the items in T. In the latter case, that is, $I > J$, the first clause applies and rewrites the call into `[`J`|insert(`I`,`S`)]`. The complete induction hypothesis holds for the call `insert(`I`,`S`)` because S has length $n-1$. (This is why we needed the induction to be complete: the hypothesis has to apply to the case $n-1$. The correctness proof on `isort/1` alone does not require it.) Therefore, its value is the correct expected sorted list of length n. Since $I > J$ and J is the smallest item of T, J has to be the smallest item of the result, which it is, being the first one in `[`J`|insert(`I`,`S`)]`. This closes the second case.

Finally, the complete induction principle entails that `insert/2` is correct, which allows us to apply the induction principle to `isort/1` and conclude that it is correct as well. QED

The keen reader may have remarked that the proof is incomplete, because we did not clearly established that the resulting list is a permutation of the original list. That is because we wanted to convey the main line of thought. Notice also how to one recursive call in the code corresponds one application of the induction hypothesis in the proof. This is not a coincidence and, on the contrary, exemplifies a duality between proof theory and programming language theory.

Exercises. (*More playful variations on two-way insertion sort*) [See answers on page 382.]

1. When designing `i2w/3`, on page 222, we chose to always push the item to be inserted, `K`, in the right list in clause ϵ. Let us modify slightly this tactic and push on the left when the left list is empty (in bold in the following definition):

```
i2w_a(L) -> i2w_a([],[],L).

i2w_a(   [],    Q,     []) -> Q;
i2w_a([I|P],    Q,     []) -> i2w_a(    P,[I|Q],[]);
i2w_a(    P,[J|Q],L=[K|_]) when J < K
```

```
                              -> i2w_a([J|P],    Q, L);
i2w_a(   [],     Q,  [K|R]) -> i2w_a(  [K],    Q, R);
i2w_a([I|P],    Q,L=[K|_]) when K < I
                              -> i2w_a(    P,[I|Q], L);
i2w_a(    P,     Q,  [K|R]) -> i2w_a(    P,[K|Q], R).
```

Find the delay in the best and worst cases for an input list of n items. Prove that the average delay is

$$\mathcal{A}_n^{\texttt{i2w\_a}} = \frac{1}{6}n^2 + \frac{3}{2}n - H_n + \frac{10}{3}.$$

From which value of n do you have $\mathcal{A}_n^{\texttt{i2w\_a}} < \mathcal{A}_n^{\texttt{i2w}}$? Can you explain in simple terms why the delay is slightly lesser?
(*Hint.* Write down the examples similar to the ones found in FIGURE 59 on page 227 and observe that the difference with `i2w/1` is that the configuration with an empty left list is replaced with a configuration with a singleton left list, that is, $\mathcal{A}_{0,k}$ is replaced with $\mathcal{A}_{1,k-1}$ in the definition of $\mathcal{A}_k$.)

2. In clause ϵ of `i2w/3`, the number `K` is inserted on the right list. Let us insert it instead on the left list (change in bold):

```
i2w_b(L) -> i2w_b([],[],L).

i2w_b(   [],     Q,     []) -> Q;
i2w_b([I|P],     Q,     []) -> i2w_b(    P,[I|Q],[]);
i2w_b(    P,[J|Q],L=[K|_]) when J < K
                              -> i2w_b([J|P],    Q, L);
i2w_b([I|P],    Q,L=[K|_]) when K < I
                              -> i2w_b(    P,[I|Q], L);
i2w_b(    P,     Q,  [K|R]) -> i2w_b([K|P],    Q, R).
```

Find the best and worst cases of `i2w_b/1`. Prove that the average delay is

$$\mathcal{A}_n^{\texttt{i2w\_b}} = \frac{1}{6}n^2 + \frac{3}{2}n + \frac{7}{3}.$$

How does that compare with `i2w/1`? Explain why.

3. When designing `i2wb/1`, we decided, in case the number of items in a configuration is odd, to always maintain the right list longer by one. What if we allowed the left list to be longer by one as well? Extend the definition of `i2wb/4` so to cope with a difference in length ranging over -1, 0 and 1. Find the best and worst cases. Find the average delay. Do you expect it to be lower? Try guessing by drawing some examples and building an argument and then do your calculations.

Chapter 11

Inductive Proofs of Programs

In this chapter, we shall use a mathematical notation for our programs because we will be dealing with mathematical proofs about them and with this typesetting will both mathematics and programs will blend in. For instance, instead of writing as usual `rev`, we will write rev; instead of `X`, we will write x. We have been using the *transitive closure* of a rewrite relationship $(\rightarrow)$, defined as $(\twoheadrightarrow) := \bigcup_{i>0} (\xrightarrow{i})$. Let $(\xrightarrow{*})$ be the reflexive-transitive closure of $(\rightarrow)$, that is, $(\xrightarrow{*}) := (=) \cup (\twoheadrightarrow)$.

Let us recall the definition

$$\mathsf{join}([\,], t) \xrightarrow{\alpha} t; \qquad \mathsf{join}([x\,|\,s], t) \xrightarrow{\beta} [x\,|\,\mathsf{join}(s,t)].$$

Let us notice that we have

$$\mathsf{join}([1],[2,3,4]) \twoheadrightarrow [1,2,3,4] \twoheadleftarrow \mathsf{join}([1,2],[3,4]).$$

It is enlightening to create *equivalence classes* of terms that are joinable: by definition, we have $a \equiv b$ if there exists a value v such that $a \xrightarrow{*} v$ and $b \xrightarrow{*} v$. For instance, $\mathsf{join}([1,2],[3,4]) \equiv \mathsf{join}([1],[2,3,4])$. If we want to prove equivalences on terms with variables ranging over infinite sets, like $\mathsf{join}(s, \mathsf{join}(t,u)) \equiv \mathsf{join}(\mathsf{join}(s,t),u)$, we need to rely on some induction principle. In this section, we explain some inductive techniques commonly used to prove desired properties of programs, like equivalence, correctness and termination.

Induction. We define a *well-founded relation* on a set A as being a binary relation $(\prec)$ which does not have any *infinite descending chains*, that is, no $\cdots \prec x_1 \prec x_0$. The *well-founded induction* then states that, for any property $\aleph$, proving $\forall x \in A.\aleph(x)$ is equivalent to proving $\forall x.(\forall y.y \prec x \Rightarrow \aleph(y)) \Rightarrow \aleph(x)$. Because there are no infinite descending chains, any subset $B \subseteq A$ contains minimal elements $M \subseteq B$, that is, there is no $y \in B$ such that $y \prec x$ if $x \in M$. In this case, well-founded induction degenerates into proving $\aleph(x)$ for all $x \in M$. When $A = \mathbb{N}$, this principle is called *mathematical (complete) induction. Structural* *induction* is another particular case where $x \prec y$ holds if, and only if, x is a proper subterm of y. For instance, we can define $x \prec [h \mid x]$, for any term h and any stack x. There is no infinite descending chain because $[\,]$ is the unique minimal element of A: $[\,] \prec x$, for all stacks x; hence the basis arises only when $x = [\,]$ and the proposition to establish degenerates into $\aleph([\,])$.

Associativity of stack catenation. Let us illustrate this principle by proving the associativity of join. Let us call $\mathcal{P}(s, t, u)$ the property "$\mathsf{join}(s, \mathsf{join}(t, u)) \equiv \mathsf{join}(\mathsf{join}(s, t), u)$." Let T be the set of all possible terms and $S \subseteq T$ be the set of all stacks. If not otherwise mentioned, we assume that a variable belongs to S. We must prove $\forall t, u.\mathcal{P}([\,], t, u)$ and $\forall s, t, u.(\mathcal{P}(s, t, u) \Rightarrow \forall x \in T.\mathcal{P}([x \,|\, s], t, u))$. The first is straightforward: $\mathsf{join}([\,], \mathsf{join}(s, t)) \xrightarrow{\alpha} \mathsf{join}(s, t) \xleftarrow{\alpha} \mathsf{join}(\mathsf{join}([\,], t), u)$. Let us assume now $\mathcal{P}(s, t, u)$, called the *induction hypothesis*, and prove $\mathcal{P}([x \mid s], t, u)$, for any term x. We have $\mathsf{join}([x \,|\, s], \mathsf{join}(t, u)) \xrightarrow{\beta} [x \,|\, \mathsf{join}(s, \mathsf{join}(t, u))] \equiv [x \,|\, \mathsf{join}(\mathsf{join}(s, t), u)] \xleftarrow{\beta} \mathsf{join}([x \,|\, \mathsf{join}(s, t)], u) \xleftarrow{\beta} \mathsf{join}(\mathsf{join}([x \,|\, s], t), u)$, therefore $\mathcal{P}([x \mid s], t, u)$ holds and, by the well-founded induction principle, $\forall s, t, u.\mathcal{P}(s, t, u)$.

Involution of stack reversal. Sometimes, a proof requires some lemma to be devised. Let us consider the definition of a function rev_0 reversing a stack:

$$\mathsf{rev}_0([\,]) \xrightarrow{\gamma} [\,]; \qquad \mathsf{rev}_0([x \,|\, s]) \xrightarrow{\delta} \mathsf{join}(\mathsf{rev}_0(s), [x]).$$

Let $\mathcal{Q}(s)$ be the property "$\mathsf{rev}_0(\mathsf{rev}_0(s)) \equiv s$." Proving the basis is immediate: $\mathsf{rev}_0(\mathsf{rev}_0([\,])) \xrightarrow{\gamma} \mathsf{rev}_0([\,]) \xrightarrow{\gamma} [\,]$. The induction hypothesis is $\mathcal{Q}(s)$ and we want to establish $\mathcal{Q}([x \,|\, s])$, for any x. If we start head-on with $\mathsf{rev}_0(\mathsf{rev}_0([x \mid s])) \xrightarrow{\delta} \mathsf{rev}_0(\mathsf{join}(\mathsf{rev}_0(s), [x]))$, we are stuck. But the term to rewrite involves both rev_0 and join, hence spurring us to conceive a lemma where the stumbling pattern $\mathsf{join}(\mathsf{rev}_0(\dots), \dots)$ occurs and is equivalent to a simpler term. Let $\mathcal{R}(s, t)$ be the property "$\mathsf{join}(\mathsf{rev}_0(t), \mathsf{rev}_0(s)) \equiv \mathsf{rev}_0(\mathsf{join}(s, t))$." In order to prove it by induction on the structure of s, we need, for all t, to draw $\mathcal{R}([\,], t)$, on the one hand, and $\mathcal{R}([x \,|\, s], t)$ if $\mathcal{R}(s, t)$, on the other hand. The former is almost within reach: $\mathsf{rev}_0(\mathsf{join}([\,], t)) \xrightarrow{\alpha} \mathsf{rev}_0(t) \leftrightsquigarrow \mathsf{join}(\mathsf{rev}_0(t), [\,]) \xleftarrow{\gamma} \mathsf{join}(\mathsf{rev}_0(t), \mathsf{rev}_0([\,]))$. The missing part is filled by showing that $(\leftrightsquigarrow)$ is $(\twoheadleftarrow)$. Let $\mathcal{S}(s)$ be the property "$\mathsf{join}(s, [\,]) \twoheadrightarrow s$." In order to prove it by induction on the structure of s, we have to prove the basis $\mathsf{join}([\,], [\,]) \twoheadrightarrow [\,]$ and $\mathcal{S}(s) \Rightarrow \forall x \in T.\mathcal{S}([x \mid s])$. The former is easy: $\mathsf{join}([\,], [\,]) \xrightarrow{\alpha} [\,]$. The latter is not complicated either: $\mathsf{join}([x \,|\, s], [\,]) \xrightarrow{\beta} [x \,|\, \mathsf{join}(s, [\,])] \twoheadrightarrow [x \,|\, s]$, where the last step makes use of the induction hypothesis $\mathcal{S}(s)$. We have now completed our proof of $\mathcal{R}([\,], t)$. Assuming $\mathcal{R}(s,t)$, we now must prove $\forall x \in T.\mathcal{R}([x\,|\,s],t)$:

$$\begin{aligned}
&\mathsf{join}(\mathsf{rev}_0(t),\mathsf{rev}_0([x\,|\,s]))\\
&\qquad\xrightarrow{\delta} \mathsf{join}(\mathsf{rev}_0(t),\mathsf{join}(\mathsf{rev}_0(s),[x]))\\
&\qquad\equiv \mathsf{join}(\mathsf{join}(\mathsf{rev}_0(t),\mathsf{rev}_0(s)),[x]) &&\text{(instance of } \mathcal{P})\\
&\qquad\equiv \mathsf{join}(\mathsf{rev}_0(\mathsf{join}(s,t)),[x]) &&\text{(hypothesis } \mathcal{R}(s,t))\\
&\qquad\xleftarrow{\delta} \mathsf{rev}_0([x\,|\,\mathsf{join}(s,t)])\\
&\qquad\xleftarrow{\beta} \mathsf{rev}_0(\mathsf{join}([x\,|\,s],t)).
\end{aligned}$$

This proves $\mathcal{R}([x\,|\,s],t)$. We now resume the proof of $\mathcal{Q}([x\,|\,s])$, assuming $\mathcal{Q}(s)$:

$$\begin{aligned}
&\mathsf{rev}_0(\mathsf{rev}_0([x\,|\,s]))\\
&\quad\xleftarrow{\alpha} \mathsf{rev}_0(\mathsf{rev}_0([x\,|\,\mathsf{join}([\,],s)]))\\
&\quad\xleftarrow{\beta} \mathsf{rev}_0(\mathsf{rev}_0(\mathsf{join}([x],s)))\\
&\quad\equiv \mathsf{rev}_0(\mathsf{join}(\mathsf{rev}_0(s),\mathsf{rev}_0([x]))) &&(\mathcal{R}([x],s))\\
&\quad\equiv \mathsf{join}(\mathsf{rev}_0(\mathsf{rev}_0([x])),\mathsf{rev}_0(\mathsf{rev}_0(s))) &&(\mathcal{R}(\mathsf{rev}_0(s),\mathsf{rev}_0([x])))\\
&\quad\equiv \mathsf{join}(\mathsf{rev}_0(\mathsf{rev}_0([x])),s) &&\text{(hypothesis } \mathcal{Q}(s))\\
&\quad\xrightarrow{\delta} \mathsf{join}(\mathsf{rev}_0(\mathsf{join}(\mathsf{rev}_0([\,]),[x])),s)\\
&\quad\xrightarrow{\gamma} \mathsf{join}(\mathsf{rev}_0(\mathsf{join}([\,],[x])),s)\\
&\quad\xrightarrow{\alpha} \mathsf{join}(\mathsf{rev}_0([x]),s)\\
&\quad\xrightarrow{\delta} \mathsf{join}(\mathsf{join}(\mathsf{rev}_0([\,]),[x]),s)\\
&\quad\xrightarrow{\gamma} \mathsf{join}(\mathsf{join}([\,],[x]),s)\\
&\quad\xrightarrow{\alpha} \mathsf{join}([x],s) \xrightarrow{\beta} [x\,|\,\mathsf{join}([\,],s)] \xrightarrow{\alpha} [x\,|\,s].
\end{aligned}$$

Equivalence of programs. We may have two definitions for the same function, which differ in complexity and/or efficiency. For instance, rev_0 was given an intuitive definition, as we can clearly see, in clause δ, that the item x, which is the top of the input, is intended to occur at the bottom of the output. Unfortunately, this definition is computationally inefficient, that is, it leads to a great deal of rewrites relatively to the size of the input. (We will give a formal meaning to this statement in a coming section.) Let us assume that we also have an efficient definition for the stack reversal, named rev, whose definition is

$$\mathsf{rev}(s) \xrightarrow{\epsilon} \mathsf{rc}(s,[\,]). \qquad \mathsf{rc}([\,],t) \xrightarrow{\zeta} t; \qquad \mathsf{rc}([x\,|\,s],t) \xrightarrow{\eta} \mathsf{rc}(s,[x\,|\,t]).$$

Let us prove the equivalence property $\mathcal{T}(s)$ stated as "$\mathsf{rev}_0(s) \equiv \mathsf{rev}(s)$." Using structural induction, we first have to prove $\mathcal{T}([\,])$ and then $\forall x \in T.\mathcal{T}([x \mid s])$, assuming $\mathcal{T}(s)$. The former is easy: $\mathsf{rev}_0([\,]) \xrightarrow{\gamma} [\,] \xleftarrow{\zeta} \mathsf{rc}([\,],[\,]) \xleftarrow{\epsilon} \mathsf{rev}([\,])$. As for the latter: $\mathsf{rev}_0([x\,|\,s]) \xrightarrow{\delta} \mathsf{join}(\mathsf{rev}_0(s),[x]) \equiv \mathsf{join}(\mathsf{rev}(s),[x]) \leftrightsquigarrow \mathsf{rc}(s,[x]) \xleftarrow{\eta} \mathsf{rc}([x\,|\,s],[\,]) \xleftarrow{\epsilon} \mathsf{rev}([x\,|\,s])$. The missing part is filled by showing ($\leftrightsquigarrow$) to be ($\equiv$). Let $\mathcal{U}(s,t)$ be the property "$\mathsf{rc}(s,t) \equiv \mathsf{join}(\mathsf{rev}(s),t)$." Induction on the structure of s yields the basis $\mathcal{U}([\,],t)$ and the general case $\mathcal{U}(s,t) \Rightarrow \forall x \in T.\mathcal{U}([x \mid s],t)$. The former is proved by $\mathsf{rc}([\,],t) \xrightarrow{\zeta} t \xleftarrow{\alpha} \mathsf{join}([\,],t) \xleftarrow{\zeta} \mathsf{join}(\mathsf{rc}([\,],[\,]),t) \xleftarrow{\epsilon} \mathsf{join}(\mathsf{rev}([\,]),t)$. The latter unfolds as follows:

$$
\begin{array}{lll}
\mathsf{rc}([x\,|\,s],t) & & \\
 & \xrightarrow{\eta} \mathsf{rc}(s,[x\,|\,t]) & \\
 & \equiv \mathsf{join}(\mathsf{rev}(s),[x\,|\,t]) & (\text{hypothesis } \mathcal{U}(s,[x\,|\,t])) \\
 & \xleftarrow{\alpha} \mathsf{join}(\mathsf{rev}(s),[x\,|\,\mathsf{join}([\,],t)]) & \\
 & \xleftarrow{\beta} \mathsf{join}(\mathsf{rev}(s),\mathsf{join}([x],t)) & \\
 & \equiv \mathsf{join}(\mathsf{join}(\mathsf{rev}(s),[x]),t) & (\text{associativity } \mathcal{P}(\mathsf{rev}(s),[x],t)) \\
 & \equiv \mathsf{join}(\mathsf{rc}(s,[x]),t) & (\text{hypothesis } \mathcal{U}(s,[x])) \\
 & \xleftarrow{\eta} \mathsf{join}(\mathsf{rc}([x\,|\,s],[\,]),t) & \\
 & \xleftarrow{\epsilon} \mathsf{join}(\mathsf{rev}([x\,|\,s]),t). &
\end{array}
$$

Termination of Ackermann's function. Let $m,n \geqslant 0$ and consider

$$
\begin{aligned}
\mathsf{ack}(0,n) &\xrightarrow{\theta} n+1;\\
\mathsf{ack}(m+1,0) &\xrightarrow{\iota} \mathsf{ack}(m,1);\\
\mathsf{ack}(m+1,n+1) &\xrightarrow{\kappa} \mathsf{ack}(m,\mathsf{ack}(m+1,n)).
\end{aligned}
$$

This is a simplified form of Ackermann's function, an early example of a total computable function which is not primitive recursive. It makes use of double recursion and two parameters to grow as a tower of exponents, for example, $\mathsf{ack}(4,3) \twoheadrightarrow 2^{2^{65536}}-3$. Our interest is that it is not obviously terminating, because if the first argument decreases, the second is allowed to increase largely. In order to deal with this definition, we need to define a well-founded relation on pairs, called *lexicographic order*. Let $(\prec_A)$ and $(\prec_B)$ be well-founded relations on the sets A and B. Then $(\prec_{A\times B})$ defined as follows on $A \times B$ is well-founded:

$$(a_0,b_0) \prec_{A\times B} (a_1,b_1) \Leftrightarrow \; a_0 \prec_A a_1 \text{ or } a_0 = a_1 \text{ and } b_0 \prec_B b_1.$$

If $A = B = \mathbb{N}$ then $(\prec_A) = (\prec_B) = (<)$. In order to prove that $\mathsf{ack}(m,n)$ terminates for all $m,n \geqslant 0$, well-founded induction requires proof that $\mathsf{ack}(0,0)$ terminates and that $\mathsf{ack}(p,q)$ terminates if $\mathsf{ack}(m,n)$ terminates for all $(m,n) \prec (p,q)$. The former is direct: $\mathsf{ack}(0,0) \xrightarrow{\theta} 1$. The latter is twofold. First, if $p > 0$ and $q = 0$, then $\mathsf{ack}(p,q) \xrightarrow{\iota} \mathsf{ack}(p-1,1)$. Lexicographically, $(p-1,1) \prec (p,q)$, thus, by the induction hypothesis, $\mathsf{ack}(p-1,1)$ terminates and so $\mathsf{ack}(p,q)$ by ι. Second, if $p > 0$ and $q > 0$, we have $\mathsf{ack}(p,q) \xrightarrow{\kappa} \mathsf{ack}(p-1,\mathsf{ack}(p,q-1))$. Because $(p,q-1) \prec (p,q)$, the induction hypothesis implies that

$\mathsf{ack}(p, q-1)$ terminates. Moreover, $(p-1, r) \prec (p, q)$, for any r, thus the same hypothesis proves that $\mathsf{ack}(p-1, r)$ terminates, in particular whence $r = \mathsf{ack}(p, q-1)$, which we just showed to be defined. Thus, by clause κ, we draw the termination of $\mathsf{ack}(p, q)$.

Correctness of insertion sort. In chapter 10 on page 238, we presented a sorting algorithm called *insertion sort*, which consists in inserting objects orderly and one by one in a stack originally empty. The traditional analogy is that of sorting a hand in a card game: from left to right, each card is moved leftward until it reaches its place (the leftmost stays put). We also introduced a proof of its correctness, but it was semi-formal, as it relied a lot on English. In this section, we want to propose a more formal correctness proof.

A total order $(\preccurlyeq^t)$ is a binary relation that is antisymmetric $(x \preccurlyeq^t y \wedge y \preccurlyeq^t x \Rightarrow x = y)$, transitive $(x \preccurlyeq^t y \wedge y \preccurlyeq^t z \Rightarrow x \preccurlyeq^t z)$ and total $(x \preccurlyeq^t y \vee y \preccurlyeq^t x)$. In particular, it is reflexive. As an example, let us restrict ourselves to sorting positive integers in nondecreasing order using $(\leqslant)$. Let $\mathsf{ins}(x, s)$ be the stack resulting from the insertion of x into the stack s, without an explicit comparison:

$$\mathsf{ins}(x, [\,]) \rightarrow [x]; \qquad \mathsf{ins}(x, [y \,|\, s]) \rightarrow [\mathsf{min}(x, y) \,|\, \mathsf{ins}(\mathsf{max}(x, y), s)].$$

We need to provide definitions to compute the minimum and maximum:

$$\begin{array}{ll}
\mathsf{max}(0, y) \rightarrow y; & \mathsf{min}(0, y) \rightarrow 0; \\
\mathsf{max}(x, 0) \rightarrow x; & \mathsf{min}(x, 0) \rightarrow 0; \\
\mathsf{max}(x, y) \rightarrow 1 + \mathsf{max}(x-1, y-1). & \mathsf{min}(x, y) \rightarrow 1 + \mathsf{min}(x-1, y-1).
\end{array}$$

While this approach fits our framework, it is both inefficient and bulky, hence it is worth extending our language so rewrite rules are selected by pattern matching only if some optional associated comparison holds. We can redefine ins as

$$\begin{array}{rlr}
\mathsf{ins}(x, [y \,|\, s]) \xrightarrow{\lambda} [y \,|\, \mathsf{ins}(x, s)], \text{ if } y \prec^t x; & \mathsf{sort}([\,]) \xrightarrow{\nu} [\,]; \\
\mathsf{ins}(x, s) \xrightarrow{\mu} [x \,|\, s]. & \mathsf{sort}([x \,|\, s]) \xrightarrow{\xi} \mathsf{ins}(x, \mathsf{sort}(s)).
\end{array}$$

Note that we used $y \prec^t x$, meaning $\neg(x \preccurlyeq^t y)$. Using $(<)$ on integers instead of $(\prec^t)$, this abstract program is translated straightforwardly in Erlang as follows:

```
ins(X,[Y|S]) when Y < X -> [Y|ins(X,S)];
ins(X,    S)            -> [X|S].

sort(   []) -> [];
sort([X|S]) -> ins(X,sort(S)).
```

It is of the utmost importance to prove the *correctness* of an important program. In fact, the concept of correctness is a relationship, so we always ought to speak of the correctness of a program *with respect to its specification.* A specification is a logical description of the expected properties of the output of a program, given some assumptions on its input. In the case of insertion sort, we need to express those properties, first informally, then formally. We would say that “The stack $\textsf{sort}(s)$ is totally ordered nondecreasingly and it contains all the items in s, but no more.” This captures all what is expected from insertion sort. Let us name $\mathcal{V}(s)$ the proposition “The stack s is sorted nondecreasingly” and $p \approx q$ the proposition “The stacks p is a permutation of the stack q,” with the provision that a permutation consists in exchanging pairs of items. To define formally these concepts, we use *inductive logic definitions*, that is, we propose a set of implications that construct objects from strictly smaller objects, according to a well-founded relation. In the present case, the relation obeys $x \prec [x\,|\,s]$ and $\mathcal{V}$ is

$$(S_0)\;\; \mathcal{V}([\,]); \quad (S_1)\;\; \mathcal{V}([x]); \quad (S_2)\;\; x \preccurlyeq^t y \Rightarrow \mathcal{V}([y\,|\,s]) \Rightarrow \mathcal{V}([x,y\,|\,s]).$$

Note that, if not otherwise mentioned, freely occurring variables are implicitly universally quantified at the level of the formula they occur in, for example, we should interpret S_1 as $\forall x \in T.\mathcal{V}([x])$. The propositions S_0 and S_1 are *axioms* because they are implied by any statement, so the symbol $(\Rightarrow)$ can be omitted. The case S_2 can be equivalently read as $(x \preccurlyeq^t y \wedge \mathcal{V}([y\,|\,s])) \Rightarrow \mathcal{V}([x,y\,|\,s])$. Moreover, since the implication is definitional, it is also an equivalence, so S_2 can also be understood as $(x \preccurlyeq^t y \wedge \mathcal{V}([y\,|\,s])) \Leftarrow \mathcal{V}([x,y\,|\,s])$. This view of an inductive definition is called an *inversion lemma.*

The relation between permutations of a stack can be defined as follows:

$$\begin{array}{ll}(P_0)\;\; [\,] \approx [\,]; & (P_2)\;\; s \approx t \Rightarrow [x\,|\,s] \approx [x\,|\,t];\\ (P_1)\;\; [x,y\,|\,s] \approx [y,x\,|\,s]; & (P_3)\;\; s \approx u \Rightarrow u \approx t \Rightarrow s \approx t.\end{array}$$

We recognise P_0 and P_1 as axioms and P_3 as transitivity. The underlying idea consists in defining a permutation as a series of *transpositions* by means of P_1, that is, exchanges of adjacent items. This approach is likely to work here because insertion can be thought of as adding an item on top of a stack and then performing a series of transpositions until the total order is restored. As a warm-up exercise, let us prove the *reflexivity* of $(\approx)$, named $\mathcal{W}(s)$: $s \approx s$. The proof technique we use is a variation on structural induction which, instead of being applied to the object at hand, like a stack, is applied to the proof itself, considered as an object—better called *meta-object.* In the present case, the way instances of the relationship $(\approx)$ are made is by a series of logical implications amounting to a linear structure, or stack, of growing instances; therefore, it is possible to apply structural induction to it. Let us demonstrate it by proving $\mathcal{W}(s)$. If P_0 produced $s \approx s$, then $s = [\,]$ and $\mathcal{W}(s)$ hold. Axiom P_1 is not reflexive unless $x = y$. Let us assume now that $\mathcal{W}$ holds for the antecedents in P_2 and P_3 (inductive hypothesis) and let us prove that it also holds for the consequents. This is quick: assuming $s = t$ in P_2 leads to $[x \mid s] \approx [x \mid s]$ and, assuming $s = u = t$ in P_3 leads to $s \approx s$.

Let us prove now the *symmetry* of $(\approx)$ by the same technique, called *induction on the structure of the proof.* Let $\mathcal{X}(s,t)$ denote "$s \approx t \Rightarrow t \approx s$." The axioms are clearly symmetric. The induction hypothesis is that $\mathcal{X}$ holds for the antecedents of P_2 and P_3. In the former case, we thus deduce $t \approx s$, which can be the antecedent of another instance of P_2 and imply $[x \,|\, t] \approx [x \,|\, s]$. This proves the symmetry of the consequent of P_2. In the last case, we deduce $u \approx s$ and $t \approx u$, which can be antecedents for P_3 itself and lead to $t \approx s$, achieving thereby the proof.

Let us now turn our attention to our main objective, which we may call $\mathcal{Y}(s)$: "$\mathcal{V}(\mathsf{sort}(s)) \wedge \mathsf{sort}(s) \approx s$." Let us tackle its proof by structural induction on s. The basis is $\mathcal{Y}([\,])$ and it is showed to hold because of rewrite rule ν and axioms S_0 and P_0. Let us assume now $\mathcal{Y}(s)$ and let us prove $\mathcal{Y}([x \mid s])$. We have the rewrite $\mathsf{sort}([x \mid s]) \xrightarrow{\xi} \mathsf{ins}(x, \mathsf{sort}(s))$. Since we want $\mathcal{V}(\mathsf{sort}([x \mid s]))$, assuming $\mathcal{V}(s)$, we realise that we need the lemma "$\mathcal{V}(s) \Rightarrow \forall x \in T.\mathcal{V}(\mathsf{ins}(x,s))$," which we may name $\mathcal{I}(s)$. Similarly, let us assume the lemma $\mathcal{J}(s)$ stating that "$\mathsf{ins}(x,s) \approx [x \,|\, s]$." In particular, $\mathcal{J}(\mathsf{sort}(s))$ is $\mathsf{ins}(x, \mathsf{sort}(s)) \approx [x \mid \mathsf{sort}(s)]$. The case P_2 and the hypothesis $\mathsf{sort}(s) \approx s$ imply $[x \mid \mathsf{sort}(s)] \approx [x \mid s]$. By the transitivity of $(\approx)$, we draw $\mathsf{ins}(x, \mathsf{sort}(s)) \approx [x \,|\, s]$, hence, backtracking, $\mathcal{Y}([x \,|\, s])$ holds and, by the induction principle, the correctness property $\forall s \in S.\mathcal{Y}(s)$.

To partially complete the previous proof, let us establish the lemma $\mathcal{J}(s)$ by structural induction on s. The basis is $\mathcal{J}([\,])$ and stands since $\mathsf{ins}(x, [\,]) \xrightarrow{\mu} [x] \approx [x]$, by P_0 and P_2. Let us assume now $\mathcal{J}(s)$ and try to deduce $\mathcal{J}([x \mid s])$. In other words, let us prove $\mathsf{ins}(x,s) \approx [x \mid s] \Rightarrow \forall y \in T.\mathsf{ins}(x, [y \mid s]) \approx [x, y \mid s]$. If $x \preccurlyeq^t y$, then $\mathsf{ins}(x, [y \mid s]) \xrightarrow{\mu} [x, y \mid s] \approx [x, y \,|\, s]$ by P_1. Otherwise, $y \prec^t x$ and $\mathsf{ins}(x, [y \,|\, s]) \xrightarrow{\lambda} [y \,|\, \mathsf{ins}(x,s)]$. By the induction hypothesis and P_2, $[y \mid \mathsf{ins}(x,s)] \approx [y, x \mid s]$. By P_1, $[y, x \,|\, s] \approx [x, y \,|\, s]$. Transitivity of $(\approx)$ applied to the two last statements leads to $\mathsf{ins}(x, [y \mid s]) \approx [x, y \mid s]$. Note that we do not need to assume that s is sorted: what matters here is that ins does not lose any of the elements it inserts, but misplacement is irrelevant.

To complete the correctness proof, we must prove lemma $\mathcal{I}(s)$, that is, $\mathcal{V}(s) \Rightarrow \forall x \in T.\mathcal{V}(\mathsf{ins}(x,s))$, by induction on the structure of s. The basis $\mathcal{I}([\,])$ is easy to check: S_0 states $\mathcal{V}([\,])$; we have the rewrite $\mathsf{ins}(x,[\,]) \xrightarrow{\mu} [x]$ and S_1 implies $\mathcal{V}([x])$. Let us prove now $\mathcal{I}([x \mid s])$ assuming $\mathcal{I}(s)$. Equivalently, let us assume

$$(H_0)\ \mathcal{V}(s), \qquad (H_1)\ \mathcal{V}(\mathsf{ins}(x,s)), \qquad (H_2)\ \mathcal{V}([y \mid s]),$$

and prove $\mathcal{V}(\mathsf{ins}(x,[y \mid s]))$. If $x \preccurlyeq^t y$, then we have the rewrite $\mathsf{ins}(x,[y \mid s]) \xrightarrow{\mu} [x,y \mid s]$. Rule S_2 and hypothesis H_2 imply $\mathcal{V}([x,y \mid s])$, therefore $\mathcal{V}(\mathsf{ins}(x,[y \mid s]))$. Otherwise $y \prec^t x$ and we have $\mathsf{ins}(x,[y \mid s]) \xrightarrow{\lambda} [y \mid \mathsf{ins}(x,s)]$. Here, things get more complicated because we need to consider the structure of s. If $s = [\,]$, then $[y \mid \mathsf{ins}(y,s)] \xrightarrow{\mu} [y,x]$. Rules S_1 and S_2 imply $\mathcal{V}([y,x])$, thus $\mathcal{V}(\mathsf{ins}(x,[y \mid s]))$. Otherwise, there exists an item z and a stack t such that $s = [z \mid t]$. If $x \preccurlyeq^t z$, then $[y \mid \mathsf{ins}(x,s)] = [y \mid \mathsf{ins}(x,[z \mid t])] \xrightarrow{\mu} [y,x,z \mid t] = [y,x \mid s]$. Also, hypothesis H_0 is $\mathcal{V}([z \mid t])$, which, by means of rule S_2, implies $\mathcal{V}([x,z \mid t])$. Since $y \prec^t x$, another application of S_2 yields $\mathcal{V}([y,x,z \mid t])$, that is $\mathcal{V}([y,x \mid s])$. This and the previous rewrite mean that $\mathcal{V}([y \mid \mathsf{ins}(x,s)])$. Further backtracking until the penultimate rewrite finally yields $\mathcal{V}(x,[y \mid s])$. The last remaining case to examine is when $z \prec^t x$. Then $[y \mid \mathsf{ins}(x,s)] = [y \mid \mathsf{ins}(x,[z \mid t])] \xrightarrow{\lambda} [y,z \mid \mathsf{ins}(x,t)]$. Hypothesis S_2 is $\mathcal{V}([y,z \mid t])$, which, by means of the *inversion lemma* of rule S_2, leads to $y \preccurlyeq^t z$. By the last rewrite, hypothesis H_1 is equivalent to $\mathcal{V}([z \mid \mathsf{ins}(x,t)])$, which, with $y \preccurlyeq^t z$, enables the use of rule S_2 again, leading to $\mathcal{V}([y,z \mid \mathsf{ins}(x,t)])$. The last rewrite then allows us to say that $\mathcal{V}([y \mid \mathsf{ins}(x,s)])$, which, following backwardly the penultimate rewrite yields the conclusion $\mathcal{V}(\mathsf{ins}(x,[y \mid s]))$.

Assessment. In this section, we defined and illustrated well-founded induction in different guises through a series of proofs whose kinds and purposes are common among theoretical computer scientists: termination, equivalence and correctness. Perhaps the most striking feature of the correctness proof is its length. More precisely, two aspects may give rise to questions. First, since the program is four lines long and the specification (the S_i and P_j) consists in a total of seven cases, it may be unclear how the proof raises our confidence in the program. Second, the proof itself is rather long, which might drive us to wonder whether any error is hiding in it. The first concern can be addressed by noting that the two parts of the specification are disjoint and thus as easy to comprehend as the program. Moreover, specifications, being logical and not necessarily computable, are likely to be more abstract and composable than programs, so a larger proof may reuse them in different instances. For instance, the predicate $\mathcal{Y}$ can easily be abstracted (higher-order) over the sorting function as $\mathcal{Y}(f,s)$: "$\mathcal{V}(f(s)) \wedge f(s) \approx s$"

and thusly applies to many sorting algorithms, with the caveat that $(\approx)$ might not always be suitable. The second concern can be completely taken care of by relying on a *proof assistant*, like Coq. For instance, the formal specification of $(\approx)$ and the automatic proofs (by means of `eauto`) of its reflexivity and symmetry consists in the following script, where `x::s` stands for $[x\,|\,s]$, (`->`) is $(\Rightarrow)$ and "`perm s t`" is $s \approx t$:

```
Set Implicit Arguments.
Require Import List.
Variable A: Type.

Inductive perm: list A -> list A -> Prop :=
  P0: perm nil nil
| P1: forall x s t, perm s t -> perm (x::s) (x::t)
| P2: forall x y s, perm (x::y::s) (y::x::s)
| P3: forall s t u, perm s u -> perm u t -> perm s t.

Hint Constructors perm.

Lemma reflexion: forall s, perm s s.
Proof. induction s; eauto. Qed.

Lemma symmetry: forall s t, perm s t -> perm t s.
Proof. induction 1; eauto. Qed.
```

Chapter 12

Higher-Order Functions

Polymorphic sorting. There is an aspect of `isort/1` and `isort_tf/1` which deserves a second thought. We described on page 48 a very useful property of many Erlang functions, called *polymorphism*, which is the ability to process values of any kind in the same, uniform manner. For example, reversing a list does not depend on the nature of the items it contains—it is a purely structural algorithm. By contrast, our definitions of `isort/1` and `isort_tf/1` rely on the usage of the predefined comparison operator (`<`) in guards. This implies that all items in the list must be pairwise comparable—for example, they can be integers. But what if we want to sort other kinds of values, like lists? Consider the very practical need to sort a set of bills: each bill can be represented by a list of prices rounded to the closest integer and the set in question by a list itself; we would then want to sort by insertion the bills by nondecreasing total amounts. If we set on writing a version of `isort/1` tailored to work only with items which are lists of integers, we are duplicating code and we would have to write a different instance every time a different kind of values to be sorted presents itself. As a consequence, what is needed here is polymorphism on *function parameters*, more precisely, the possibility of a function to be a value, thus to be used as an argument. Erlang provides this facility in a natural way and many functional languages do as well. In our case, we need the caller of `isort/1` to provide an additional parameter which is a comparison function between the items. Then `isort/2` would make use of this caller-defined comparison, instead of always applying the default operator (`<`) which works only (or mostly) on integers. Here is again the definition of `isort/1`:

```
isort(   [])                 $\xrightarrow{\beta}$ [];
isort([I|L])                 $\xrightarrow{\gamma}$ insert(I,isort(L)).
insert(I,[J|S]) when I > J $\xrightarrow{\delta}$ [J|insert(I,S)];
insert(I,    S)            $\xrightarrow{\epsilon}$ [I|S].
```

Then, our first attempt at modification leads us straightforwardly to

```
isortf(_,   [])               $\xrightarrow{\beta}$ [];
isortf(F,[I|L])               $\xrightarrow{\gamma}$ insert(F,I,isortf(F,L)).
insert(F,I,[J|S]) when F(I,J) $\xrightarrow{\delta}$ [J|insert(F,I,S)];
insert(_,I,    S)             $\xrightarrow{\epsilon}$ [I|S].
```

But the Erlang compiler would reject this program because the language does not allow a user-defined function to be called in a guard. The rationale is that the call `F(I,J)` above may not terminate and the semantics, that is, the expected behaviour of Erlang constructs, is that pattern matching with guards always ends. Because it is impossible to automatically check whether a function call terminates on all inputs (this problem is equivalent to the famous *Turing halting problem*, which is *undecidable*), the compiler does not even try and prefers to reject all guards made of function calls. Thus we must move the call `F(I,J)` inside the body of the same clause, which begets the question as how to merge clauses δ and ϵ into a new clause δ_0. A simple way out is to create another function, `triage/4`, whose task is to take the result of the comparison and proceed with the rest of the computation. Of course, this means that `triage/4` must also receive all necessary information to carry on:

```
isortf(_,   [])          $\xrightarrow{\beta}$ [];
isortf(F,[I|L])          $\xrightarrow{\gamma}$ insert(F,I,isortf(F,L)).
insert(F,I,[J|S])        $\xrightarrow{\delta_0}$ triage(F,I,[J|S],F(I,J)).
triage(F,I,[J|S],$\boxed{\quad}$) $\xrightarrow{\zeta}$ [J|insert(F,I,S)];
triage(F,I,[J|S],$\boxed{\quad}$) $\xrightarrow{\eta}$ [I|S].
```

The empty boxes must be filled with the result of a comparison. In our case, we want a comparison with two possible outputs, depending on the first argument argument being lower or greater than the second. By definition, the result of `I > J` is the atom `true` if the value of `I` is greater than the value of `J` and `false` otherwise. Let us follow the same convention for `F` and impose that the value of `F(I,J)` is the atom `true` if `I` is greater than `J` and `false` otherwise. It is even better to rename the parameter `F` into something more intuitive according to its behaviour, like `Gt` (***G**reater **t**han*):

```
triage(Gt,I,[J|S],true)    $\xrightarrow{\zeta}$ [J|insert(Gt,I,S)];
triage(Gt,I,[J|S],false)   $\xrightarrow{\eta}$ [I|S].
```

We notice that clause η makes no use of `J`, which means we actually lose an item. What went wrong and when? The mistake came from not realising that clause ϵ covered two cases, `S` is empty or not, therefore we should have untangled these two cases before merging clause ϵ with clause δ, because in δ we have the pattern `[J|S]`, that is, the non-empty case. Let us rewind and split ϵ into ϵ_0 and ϵ_1:

```
isortf( _,   [])                $\xrightarrow{\beta}$ [];
isortf(Gt,[I|L])                  $\xrightarrow{\gamma}$ insert(Gt,I,isortf(Gt,L)).
insert(Gt,I,[J|S]) when Gt(I,J) $\xrightarrow{\delta}$ [J|insert(Gt,I,S)];
insert(Gt,I,[J|S])              $\xrightarrow{\epsilon_0}$ [I|[J|S]];
insert( _,I,   [])              $\xrightarrow{\epsilon_1}$ [I].
```

Notice that we did not write

```
insert(_,I,   [])               $\xrightarrow{\epsilon_1}$ [I];
insert(_,I,    S)               $\xrightarrow{\epsilon_0}$ [I|S].
```

even though it would have been correct, because we had in mind the fusion with clause δ, so we needed to make the pattern `[J|S]` conspicuous in ϵ_0. For even more clarity, we made apparent the parameter `Gt`: the heads of clauses δ and ϵ_0 are now identical and ready to merge into a new clause δ_0:

```
isortf( _,   [])          $\xrightarrow{\beta}$ [];
isortf(Gt,[I|L])          $\xrightarrow{\gamma}$ insert(Gt,I,isortf(Gt,L)).
insert(Gt,I,[J|S])        $\xrightarrow{\delta_0}$ triage(Gt,I,[J|S],Gt(I,J)).
insert( _,I,   [])        $\xrightarrow{\epsilon_1}$ [I].
triage(Gt,I,[J|S],true)   $\xrightarrow{\zeta}$ [J|insert(Gt,I,S)];
triage( _,I,[J|S],false)  $\xrightarrow{\eta}$ [I|[J|S]].
```

We can improve a little bit clause η by not distinguishing `J` and `S`:

```
triage( _,I,    S,false)  $\xrightarrow{\eta}$ [I|S].
```

This transformation is correct because `S` is never empty. Instead of using an auxiliary function like `triage/4`, which takes many arguments and serves no purpose other than performing a test on the value of `Gt(I,J)` and proceed accordingly, we can make good use of a special Erlang construct based on the new keywords `case`, `of` and `end`:

```
isortf( _,   [])   $\xrightarrow{\beta}$ [];
isortf(Gt,[I|L])   $\xrightarrow{\gamma}$ insert(Gt,I,isortf(Gt,L)).
insert(Gt,I,[J|S]) $\xrightarrow{\delta_0}$ case Gt(I,J) of
                        true  $\xrightarrow{\zeta}$ [J|insert(Gt,I,S)];
                        false $\xrightarrow{\eta}$ [I|[J|S]]
                      end;
insert( _,I,   []) $\xrightarrow{\epsilon_1}$ [I].
```

We can decrease the memory usage again in the clause η (case `false`), this time by means of an alias for the pattern `[J|S]`, so the best version of the code is

```
isortf( _,   [])      $\xrightarrow{\beta}$ [];
isortf(Gt,[I|L])      $\xrightarrow{\gamma}$ insert(Gt,I,isortf(Gt,L)).
insert(Gt,I,Q=[J|S])  $\xrightarrow{\delta_0}$ case Gt(I,J) of
                          true  $\xrightarrow{\zeta}$ [J|insert(Gt,I,S)];
                          false $\xrightarrow{\eta}$ [I|Q]
                        end;
insert( _,I,     [])  $\xrightarrow{\epsilon_1}$ [I].
```

Before we keep on studying `isortf/2` and `insert/3`, let us put to good use the `case` construct of Erlang to improve the definition of `i2wb/4`, on page 232. We always determine the delay of a function by counting the number of rewrites (in other words, the number of function calls) it requires to compute its result. Of course, this is not the whole story. For instance, comparisons do take some real time to be performed, although they are individually fast when applied to integers. Another reason why we ignored them up to now is that we always included these integer comparisons into guards, so they were not taken into account, just as the very real time taken to match an argument against a pattern in the head has been ignored all along. We may improve the legibility of our programs by factoring some matches on integers *outside the heads*. A speed-up is not guaranteed because we do not suppose anything about how the compiler processes the heads. Here is a better `isort2wb/4`:

```
isort2wb(L) -> isort2wb([],[],L,0).

isort2wb(   [],    Q,   [],_)           -> Q;
isort2wb([I|P],    Q,   [],D)           ->
                                          isort2wb(P,[I|Q],[],D);
isort2wb(    P,[J|Q],[K|R],D) when J < K ->
  case D of
    1 -> isort2wb([J|P],    ins_up(K,Q),R,0);
    0 -> isort2wb(    P,[J|ins_up(K,Q)],R,1)
  end;
isort2wb([I|P],    Q,[K|R],D) when K < I ->
  case D of
    1 -> isort2wb([I|ins_down(K,P)],    Q,R,0);
    0 -> isort2wb(    ins_down(K,P),[I|Q],R,1)
  end;
isort2wb(    P,    Q,[K|R],D)           ->
  case D of
    1 -> isort2wb([K|P],    Q,R,0);
    0 -> isort2wb(    P,[K|Q],R,1)
```

```
end.
```

Let us resume now our examination of `isortf/1` by determining its average delay. It is important *not* to count the arrows in a `case` construct because these arrows are certainly not compiled as a function call but as a conditional. Of course, the expression between the `case` and `of` keywords may contain function calls and these must be accounted for. In the function under consideration, the relevant call is `Gt(I,J)`, where the function `fun Gt/2` is a parameter. Because of this polymorphism, the delay of the call cannot be determined statically, that is, by solely examining the definition, and the delay of `isortf/2` depends on the delay of its parameter `Gt`, whose delay is a priori unknown because an infinite number of functional arguments are possible. Let us suppose that this delay is constant and let us call it $\mathcal{D}^{\mathsf{Gt}}$. The only difference, as far as delays are concerned, between `isortf/1` and `isort/1`, is the call `Gt(I,J)`. We can make an analysis similar to the one we did on page 211:

$$\mathcal{A}_0^{\mathsf{isortf}} \overset{\beta}{=} 1; \quad \mathcal{A}_{k+1}^{\mathsf{isortf}} \overset{\gamma}{=} 1 + \mathcal{A}_k^{\mathsf{insert}} + \mathcal{A}_k^{\mathsf{isortf}}, \text{ with } k \geqslant 0.$$

Telescoping both sides in a sum yields, for $n \geqslant 0$,

$$\mathcal{A}_n^{\mathsf{isortf}} = 1 + n + \sum_{k=0}^{n-1} \mathcal{A}_k^{\mathsf{insert}}.$$

The delay for using once clause δ_0 does not include any delay due to ζ nor η because these are not clauses but cases, therefore it amounts to $\mathcal{D}^{\mathsf{Gt}} + 1$. There are $k+1$ ways to insert an item into a list of k items. For an insertion before the item at location j, the head of the list being at location 0, it thus takes $(\mathcal{D}^{\mathsf{Gt}} + 1)j$ rewrites. The insertion at the end of the list, that is, at position $k+1$, is special, because the insertion, properly speaking, is performed by clause ϵ_1 instead of η, which means that `Gt` is not called for the last rewrite. So the delay for inserting after the last item is $(\mathcal{D}^{\mathsf{Gt}} + 1)k + 1$. Now we can form

$$\begin{aligned}\mathcal{A}_k^{\mathsf{insert}} &= \frac{1}{k+1}\left(\sum_{j=1}^{k}((\mathcal{D}^{\mathsf{Gt}} + 1)j) + ((\mathcal{D}^{\mathsf{Gt}} + 1)k + 1)\right)\\ &= (\mathcal{D}^{\mathsf{Gt}} + 1)\left(\frac{1}{2}k + 1\right) - \frac{\mathcal{D}^{\mathsf{Gt}}}{k+1}.\end{aligned}$$

We can check the case $\mathcal{D}^{\mathsf{Gt}} = 0$, which coincides with equation (10.12) on page 212:

$$\mathcal{A}_k^{\mathsf{insert}} = \frac{k}{2} + 1, \text{ if } \mathcal{D}^{\mathsf{Gt}} = 0.$$

Let us resume:

$$\sum_{k=0}^{n-1}\mathcal{A}_k^{\mathsf{insert}} = \frac{1}{2}(\mathcal{D}^{\mathsf{Gt}}+1)\sum_{k=0}^{n-1}k + (\mathcal{D}^{\mathsf{Gt}}+1)n - \mathcal{D}^{\mathsf{Gt}}\sum_{k=0}^{n-1}\frac{1}{k+1}$$
$$= \frac{1}{4}(\mathcal{D}^{\mathsf{Gt}}+1)n^2 + \frac{3}{4}(\mathcal{D}^{\mathsf{Gt}}+1)n - \mathcal{D}^{\mathsf{Gt}}H_n.$$

With the convention $H_0 = 0$, we finally conclude, for $n \geqslant 0$,

$$\mathcal{A}_n^{\mathsf{isortf}} = 1 + n + \left(\frac{1}{4}(\mathcal{D}^{\mathsf{Gt}}+1)n^2 + \frac{3}{4}(\mathcal{D}^{\mathsf{Gt}}+1)n - \mathcal{D}^{\mathsf{Gt}}H_n\right)$$
$$= \frac{1}{4}(\mathcal{D}^{\mathsf{Gt}}+1)n^2 + \frac{1}{4}(3\mathcal{D}^{\mathsf{Gt}}+7)n - \mathcal{D}^{\mathsf{Gt}}H_n + 1. \qquad (12.18)$$

A quick check again consists in plugging $\mathcal{D}^{\mathsf{Gt}} = 0$ into the formula and it indeed becomes identical to equation (10.13) on page 212:

$$\mathcal{A}_n^{\mathsf{isort}} = \frac{1}{4}n^2 + \frac{7}{4}n + 1, \text{ if } \mathcal{D}^{\mathsf{Gt}} = 0.$$

How would we call `isortf/2` so the resulting value is the same as calling `isort/1`? First, we need a comparison function which behaves exactly like the operator (`>`):

```
gt_int(I,J) when I > J -> true;
gt_int(_,_)            -> false.
```

If we try now to form the call

```
isortf(gt_int,[5,3,1,4,2]),
```

we find that an error occurs at run-time because `gt_int` is an atom, not a function. That is why Erlang provides a special syntax for denoting functions used as values:

```
isortf(fun gt_int/2,[5,3,1,4,2]).
```

Notice the new keyword `fun` and the usual indication of the number of arguments the function is expected to operate on (here, two).

What would happen if we passed as a parameter the function `lt_-int/2` defined as follows?

```
lt_int(I,J) when I < J -> true;
lt_int(I,J)            -> false.
```

The consequence is that the result is sorted non-increasingly and all we had to do was to change the comparison function, *not the sorting function itself.*

It may seem a burden to have to name even simple comparison functions like `lt_int/2`, which is none other than the predefined operator (`<`). Fortunately, Erlang provides a way to define functions without giving them a name. The rule consists in using the fun keyword together with the end keyword and put the usual definition in-between, without a function name. Reconsider for example the previous calls but using such anonymous functions (sometimes called *lambdas*):

```
isortf(fun(I,J) -> I > J end,[5,3,1,4,2])
```

results in [1,2,3,4,5] and

```
isortf(fun(I,J) -> I < J end,[5,3,1,4,2])
```

results in [5,4,3,2,1]. The delay for these two comparison functions is 1, so the average delay of the function call isortf(fun(I,J) -> I > J end,L), where L is a list of n integers, is obtained by setting $\mathcal{D}^{\mathsf{Gt}} = 1$ in the previously found equation (12.18) on page 258:

$$\begin{aligned}\mathcal{A}_n^{\mathsf{isortf}} &= \frac{1}{4}(\mathcal{D}^{\mathsf{Gt}}+1)n^2 + \frac{1}{4}(3\mathcal{D}^{\mathsf{Gt}}+7)n - \mathcal{D}^{\mathsf{Gt}}H_n + 1\\ &= \frac{1}{2}n^2 + \frac{5}{2}n - H_n + 1, \text{ if } \mathcal{D}^{\mathsf{Gt}} = 1.\end{aligned}$$

Let us now use isortf/2 to sort lists of lists of integers, according to the sum of the integers in each list—this is the practical application of sorting bills we previously mentioned. As the example of sorting in non-increasing order hints at, we only need here to write how to compare two lists of integers by means of the sum/1 function, whose several definitions we have already reviewed. Here is a simple one on page 30:

```
sum_tf([N|L])   -> sum_tf(L,N).
sum_tf(   [],A) -> A;
sum_tf([N|L],A) -> sum_tf(L,A+N).
```

We have $\mathcal{D}_n^{\mathsf{sum\_tf}} = n + 2$. Now we can define the comparison function gt_bill/2, based upon the operator (>):

```
gt_bill(P,Q) -> sum_tf(P) > sum_tf(Q).
```

Notice in passing that the predefined Erlang comparison operator (>) results in either the atom true or false, so there is no need to use a case construct. Then we can sort our bills by simply calling

```
isortf(fun gt_bill/2,[[1,5,2,9],[7],[2,5,11],[4,3]])
```

or, simply,

```
isortf(fun(P,Q) -> sum_tf(P) < sum_tf(Q) end,
       [[1,5,2,9],[7],[2,5,11],[4,3]]).
```

(By the way, do you expect [7] to appear before or after [4,3] in the answer? What would you have to modify so the relative order of these two lists is reversed?) It is just as easy to sort the bills in non-increasing order. This great easiness in passing around functions as any other kind of values is what justifies the adjective *functional* for a language like Erlang and many others. A function taking another function as a parameter is said to be a *higher-order function*.

Sorted association lists. There is something that we could improve in the previous definition of `isortf/2`. Sorting by comparison, as the worst case of insertion sort demonstrates eloquently, may imply that some items are compared more than once with others. It may be that the comparison has a small delay but, compounded over many uses, it leads to a significant delay. In the case of sorting bills, it is more efficient to compute all the total amounts first and then only use these amounts during the sort process, because comparing one integer to another is much faster than recomputing the sum of many integers in a list. So, what is sorted is a list of pairs whose first component, called the *key*, is a simple and small representative of the second component, called the *value* (improperly, as keys are Erlang values as well, but such is the traditional nomenclature). This data structure is sometimes called an *association list*. Only the key is used for sorting, not the value, therefore, if the key is an integer, the comparison on the key is likely to be faster than on the values. The only penalty is that all the keys must be precomputed in a first pass over the initial data and they must be stripped from the final result in an additional postprocessing. This time we shall design these first and last passes in the most general fashion by parameterisation upon the computation `Mk` of the keys:

```
% Preprocessing
mk_keys( _,        [])  -> [];
mk_keys(Mk,[V|Values])  -> [{Mk(V),V}|mk_keys(Mk,Values)].

% Postprocessing
rm_keys(            []) -> [];
rm_keys([{_,V}|KeyVal]) -> [V|rm_keys(KeyVal)].
```

The delay of `mk_keys/2` depends on the delay of `Mk`. The delay of `rm_-keys(`L`)` is $n+1$ if L contains n pairs key-value. Now we can sort by calling `isortf/2` with a comparison on two keys and with the function to build the keys, `sum_tf/1`. For instance:

```
 rm_keys(isortf(fun({K1,_},{K2,_}) -> K1 > K2 end,
                mk_keys(fun sum_tf/1,
                        [[1,5,2,9],[7],[2,5,11],[4,3]])))
```

It is very important to notice that we did not need to redefine `isortf/2`. Actually, `isortf/2`, `mk_keys/2` and `rm_keys/1` would very well constitute a library by grouping their definitions in the same module. The client, that is, the user of the library, would then provide the comparison function fitted to her data to be sorted and the function making the keys. This *modularisation* is enabled by polymorphism and higher-order functions. Finally, let us determine the general average delay of

```
rm_keys(isortf(fun({K1,_},{K2,_}) -> K1 > K2 end,
               mk_keys(fun sum_tf/1,L)))
```

where $L = [L_0, L_1, \ldots, L_{n-1}]$ is a list of n lists, each of them containing l_i integers and, in total, p integers. Since the call is the composition of several calls, let us decompose it as the sum of the following delays:

$$\mathcal{D}^{\texttt{mk\_keys}}_{n,p} + \mathcal{A}^{\texttt{isortf}}_{n} + \mathcal{D}^{\texttt{rm\_keys}}_{n}.$$

Furthermore, we have

$$\mathcal{D}^{\texttt{mk\_keys}}_{n,p} = (n+1) + \sum_{i=0}^{n-1}(\mathcal{D}^{\texttt{sum\_tf}}_{l_i} + 2) = 3n + p + 1,$$

$$\mathcal{A}^{\texttt{isortf}}_{n} = \frac{1}{2}n^2 + \frac{5}{2}n - H_n + 1, \quad \mathcal{D}^{\texttt{rm\_keys}}_{n} = n + 1.$$

Therefore, the total average delay is

$$\frac{1}{2}n^2 + \frac{13}{2}n - H_n + p + 3.$$

This delay is quadratic in the number of sub-lists but linear in the total number of integers, so the *maximum average delay* (sometimes called *max-mean delay*) happens, for a fixed value of p, when n is maximum, that is, when $n = p$—in other words, when each sub-list is a singleton. Dually, the *minimum average delay* (called *min-mean delay*) happens when $n = 1$, that is, when there is only one sub-list.

As a last example proving the versatility of our program, let us sort lists by their non-increasing lengths:

```
rm_keys(isortf(fun({K1,_},{K2,_}) -> K1 < K2 end,
               mk_keys(fun len_tf/1,
                       [[1,5,2,9],[7],[2,5,11],[4,3]])))
```

where `len_tf/1` is defined as

```
len_tf(L)    -> len(L,0).
len(   [],N) -> N;
len([_|L],N) -> len(L,N+1).
```

The result is `[[1,5,2,9],[2,5,11],[4,3],[7]]`. Notice that `[4,3]` occurs before `[7]` because the former is longer.

Let us specialise further `isortf/1`. Here is the definition again:

```
isortf( _,   [])      $\xrightarrow{\beta}$ [];
isortf(Gt,[I|L])      $\xrightarrow{\gamma}$ insert(Gt,I,isortf(Gt,L)).
insert(Gt,I,Q=[J|S]) $\xrightarrow{\delta_0}$ case Gt(I,J) of
                           true  $\xrightarrow{\zeta}$ [J|insert(Gt,I,S)];
                           false $\xrightarrow{\eta}$ [I|Q]
                         end;
insert( _,I,     []) $\xrightarrow{\epsilon_1}$ [I].
```

It is clear that if items are repeated in the input list, the call `Gt(I,J)` is expected to be rewritten to `false` at least once, therefore duplicates are kept by clause η and their relative order is preserved, that is, the sorting algorithm is *stable*. What if we do not want to preserve such duplicates in the output? We need to rewrite the definition to support this choice. The choice itself, that is, to keep or not to keep, would naturally be implemented as an additional functional parameter, say `Eq`. Also, we would need to change our assumption about the values of the comparison: we need three cases, so the equality case is explicit. Let us modify the variable `Gt` to reflect this increase in detail and call it more generally `Cmp` (*compare*). The corresponding arguments should be values amongst the user-defined axioms `lt` (*lower than*), `gt` (*greater than*) and `eq` (*equal*). We have

```
isortf(  _, _,   [])      $\xrightarrow{\beta}$ [];
isortf(Cmp,Eq,[I|L])      $\xrightarrow{\gamma}$
  insert(Cmp,Eq,I,isortf(Cmp,Eq,L)).
insert(Cmp,Eq,I,Q=[J|S]) $\xrightarrow{\delta_0}$
  case Cmp(I,J) of
    gt  $\xrightarrow{\zeta}$ [J|insert(Cmp,Eq,I,S)];
    lt  $\xrightarrow{\eta}$ [I|Q];
    eq  $\xrightarrow{\theta}$ Eq(I,Q)                                % New case
  end;
insert(  _, _,I,     []) $\xrightarrow{\epsilon_1}$ [I].
```

Now, let us say that we want to sort nondecreasingly a list of integers and retain possible redundant numbers, just as the previous version allowed. We have (novelty in bold):

```
isortf(fun(I,J) -> I>J end,fun(I,Q) -> [I|Q] end,[5,3,1,4])
```

which results in `[1,3,4,5]`. If we do not want the numbers repeated, we form instead the call

```
isortf(fun(I,J) -> I>J end,fun(_,Q) ->     Q end,[5,3,1,4])
```

resulting in `[1,3,4,5]`. In passing, this technique allows us to solve the problem of removing duplicates in a list of items for which there is a total order. However, if only successive duplicates have to be removed from a list, the function `compress/1` on page 168:

```
compress(         []) -> [];
compress([I|L=[I|_]]) -> compress(L);
compress(      [I|L]) -> [I|compress(L)].
```

is more efficient because its growth is linear in the input size.

We would be remiss if we do not mention that a higher-order function is not only a function whose at least one parameter is a function, but it also can be a function whose calls evaluate in a function. This kind of function is said to be *curried*, as an homage to the logician Haskell Curry. The possibility was already there when we introduced the keywords `fun` and `end`, because they allow us to define an anonymous function and use it just like another value, so nothing impeded us from using such a functional value as the result of a named function, like in the following function mathematically composing two functions:

```
compose(F,G) -> (fun(X) -> F(G(X)) end).
```

Actually, the parentheses around the functional value are useless if we remember that the keywords `fun` and `end` play the role of parentheses when the anonymous function is not called:

```
compose(F,G) -> fun(X) -> F(G(X)) end.
```

The higher-order function `compose/2` can be used to compute the composition of two other functions, the result being a function, of course.

Maps. We may want to compute a function which sums the images of a list L of integers by a given function f. In mathematical notation, the final result would be expressed as

$$\sum_{k \in L} f(k).$$

In order to implement this in Erlang, we must proceed in two steps: firstly, we need a higher-order function which computes the images of the items of a list by a function; secondly, we need a function summing the integers of a list. We already have the latter, known from page 259 as `sum_tf/1`. The former is traditionally called `map/2`, such that the call `map(`F`,`L`)` applies function F to all the items of list L and evaluates into the list of the results. That is,

$$\texttt{map(}F\texttt{,[}I_0\texttt{,}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{])} \equiv \texttt{[}F\texttt{(}I_0\texttt{),}F\texttt{(}I_1\texttt{),}\ldots\texttt{,}F\texttt{(}I_{n-1}\texttt{)].}$$

With this goal in mind, it is straightforward to define `map/2`:

```
map(_,   []) -> [];
map(F,[I|L]) -> [F(I)|map(F,L)].
```

The function we were looking for is now compactly defined as the composition of map/2 and sum_tf/1 as follows:

```
sum_f(F) -> fun(L) -> sum_tf(map(F,L)) end.
```

For instance, the function call

```
sum_f(fun(X) -> X*X end)
```

denotes the function which sums the squares of the numbers in a list to be provided. It is equivalent to the value

```
fun(L) -> sum_tf(map(fun(X) -> X*X end,L)) end.
```

It is possible to call this function just after is has been computed, but *parentheses must be added around a function being called when it is anonymous*. For instance, see the bold typefaces in

```
(sum_f(fun(X) -> X*X end))([1,2,3]).
```

The function map/2 is often used because it captures a common operation on lists. For example, on page 197, we defined

```
push(_,          []) -> [];
push(I,[Perm|Perms]) -> [[I|Perm]|push(I,Perms)].
```

It is equivalent to

```
push(I,Perms) -> map(fun(Perm) -> [I|Perm] end,Perms).
```

This style leads to clearer programs as it shows the underlying recursive computation without having to read or write a definition for it. In other words, using a higher-order function like map/2 allows us to identify a common recursive pattern and let the programmer focus instead on the specific processing of the items. We shall encounter other examples in the next sections but, before we move on, imagine we typed instead

```
push(I,Perms) -> map(fun(Perms) -> [I|Perms] end,Perms).
```

The Erlang compiler would issue the following warning:

```
Warning: variable 'Perms' shadowed in 'fun'.
```

What happens is that the parameter **Perms** (in bold) "hides" the parameter Perms of push/2 in the sense that, in the body of the anonymous function, any occurrence of Perms refers to **Perms**. In this case, it is not an error, but the compiler designers worried about programmers walking on the shadowy side of the street. For example,

```
push(I,Perms) -> map(fun(I) -> [I|I] end,Perms).       % Capture
```

is definitely wrong because the two variables I in [I|I], which is the body of the anonymous function, are the parameter of the anonymous function. A faulty shadowing is called a *capture*. Here, the parameter I bound by `push(I,Perms)` has been captured to mean instead the parameter of `fun(I)`. As a guideline, it is best to avoid shadowing a parameter by another, as the Erlang compiler reminds us for our own sake. Note that

```
sum_tf(fun(L) -> L*L end)
```

is fine because it is equivalent to

```
fun(L) -> sum_tf(map(fun(L) -> L*L end,L)) end
```

which is a correct shadowing.

Folds. Some other useful and frequently recursive patterns can be conveniently reified into some other higher-order functions. Consider a function which traverses completely a list while processing an accumulator depending on the current visited item. In the end, the result is the final value of the accumulator, or else another function is called to finalise it. A simple example is a function computing the length of a list found on page 261:

```
len_tf(L)     -> len(L,0).
len(   [],N) -> N;
len([_|L],N) -> len(L,N+1).
```

In this case, the accumulator is an integer and the operation on it consists in incrementing it, whatever the current item is. Another function reverses a list, on page 60:

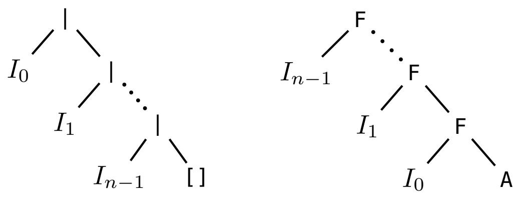
```
rev(L)             -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Here, the accumulator is a list and the operation on it consists in pushing the current item on top of it. Let us abstract separately these two concerns in a higher-order function

1. which takes as input the function creating a new accumulator from the current item and the previous accumulator and
2. which applies it successively to all the items of a parameter list.

One famous function doing exactly this is called `foldl/3` in Erlang, which stands for "fold left," because once the new accumulator for some item has been computed, the prefix of the list up to it can be folded back, as if the list were written down on a sheet of paper, because it is no longer useful. So the name should be better read as "fold from left to right" or *rightward fold*. We want

$$\texttt{foldl(}F\texttt{,}A\texttt{,[}I_0\texttt{,}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{])} \equiv F\texttt{(}I_{n-1}\texttt{,}\ldots\texttt{,}F\texttt{(}I_1\texttt{,}F\texttt{(}I_0\texttt{,}A\texttt{))}\ldots\texttt{)},$$



(a) List `L` (b) Value of `foldl(F,A,L)`

FIGURE 63: The result of `foldl/3` on a non-empty list

where A is the initial value of the accumulator. FIGURE 63 on page 266 shows the corresponding abstract syntax trees. The following definition implements the desired effect:

```
foldl(_,A,   []) -> A;
foldl(F,A,[I|L]) -> foldl(F,F(I,A),L).
```

Now we can rewrite new definitions of `len_tf/1` and `rev/1`:

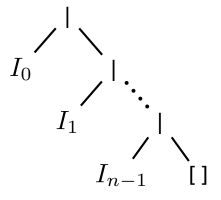
```
lenl(L) -> foldl(fun(_,A) ->   A+1 end, 0,L).
revl(L) -> foldl(fun(I,A) -> [I|A] end,[],L).
```

Function `foldl/3` is not in tail form because of the embedded call `F(I,A)`, but only a constant amount of control stack is used for the recursion of `foldl/3` itself (one node). In our two examples, `F` is in tail form, therefore these new definitions are *almost* in tail form and can stand against the originals. More definitions almost in tail form are

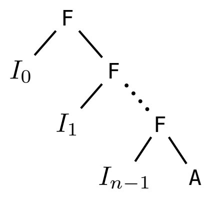
```
suml([N|L])    -> foldl(fun(I,A) ->      I+A end, N,L).
rev_joinl(P,Q) -> foldl(fun(I,A) ->    [I|A] end, Q,P).
rev_map(F,L)   -> foldl(fun(I,A) -> [F(I)|A] end,[],L).
```

Again, the reason why these definitions are not exactly in tail form is due to the call `F(I,A)` in the definition of `foldl/3`, *not* because of the functional arguments `fun(I,A) -> ... end` in the calls to `foldl/3`: they are not function calls but function definitions. The main advantage of using `foldl/3` is that it allows the programmer to focus exclusively on the processing of the accumulator, whilst `foldl/3` itself provides the ride for free. Moreover, we can easily compare different functions defined by means of `foldl/3`.

When the accumulator is a list on which values are pushed, the result is in reverse order with respect to the input. That is why `rev_map/2`, above, is not `map/2`. The former is to be preferred over the latter if the order of the items is not relevant, because `map/2` requires a control stack as long as the input list. This leads us quite naturally to introduce another higher-order function: `foldr/3`, meaning "fold from right to



(a) List `L`



(b) Value of `foldr(F,A,L)`

FIGURE 64: The result of `foldr/3` on a non-empty list

left," or *leftward fold*. We expect

$$\texttt{foldr(}F\texttt{,}A\texttt{,[}I_0\texttt{,}I_1\texttt{,}\ldots\texttt{,}I_{n-1}\texttt{])} \equiv F\texttt{(}I_0\texttt{,}F\texttt{(}I_1\texttt{,}\ldots\texttt{,}F\texttt{(}I_{n-1}\texttt{,}A\texttt{))}\ldots\texttt{)}.$$

FIGURE 64 on page 267 shows the corresponding abstract syntax trees. We achieve this behaviour with the following definition:

```
foldr(_,A,   []) -> A;
foldr(F,A,[I|L]) -> F(I,foldr(F,A,L)).
```

This definition, like `foldl/3`, is not in tail form but, unlike `foldl/3`, it requires a control stack as long as the input list. This is due to *the recursive call of `foldl/3` being in tail position*, that is, if its arguments were in tail form, the recursive call would be in tail form as well. In the case of `foldr/3`, it is the recursive call itself which is not in tail position, so, even if the functional parameter `F` were in tail form, the amount of control stack used by the call to `foldr/3` would be proportional to the length of the input list.

With the help of `foldr/3`, we can redefine `map/2` and `join/2` as

```
mapr(F,L)  -> foldr(fun(I,A) -> [F(I)|A] end,[],L).
joinr(P,Q) -> foldr(fun(I,A) ->    [I|A] end, P,Q).
```

Compare `rev_joinl/2`, defined above, with `joinr/2`: the role of the accumulator and of the input list have been exchanged, as well as `foldl/3` and `foldr/3`. It is also possible to define

```
lenr(L)     -> foldr(fun(_,A) -> 1+A end, 0,L).      % To avoid
sumr([N|L]) -> foldr(fun(I,A) -> I+A end, N,L).      % To avoid
isortr(L)   -> foldr(fun insert/2,       [],L).      % To avoid
```

but that would be unwise because `foldr/3` does not use a bounded amount of control stack, contrary to `foldl/3`. In the case of `isort/1`, it is best to call `foldl/3` instead because the order of insertion does not matter in average (although it swaps the best and worst cases if numbers are not repeated, as seen on page 217). This leads us to formulate some guidelines about the transformation into tail form. We already know that a definition in tail form is worth having or even necessary if the maximum size of the control stack is smaller than some input recursively traversed in its greatest extension. No speed-up should be expected a priori from turning a definition not in tail form into one in tail form—although this may happen sometimes. Usually,

- it is preferable, if possible, to use `foldl/3` instead of `foldr/3` because, provided the functional parameter is in tail form, the call will use a small limited amount of control stack (if the parameter is not in tail form, at least `foldl/3` won't burden further the control stack, contrary to `foldr/3`);
- when writing our own recursion, that is, without resorting to folds, it is best to have it in tail form if the accumulator is an integer, otherwise, the maximum size of the control stack may need to be proportional to the size of the input, despite the output being a single integer. (Contrast `sum/1` and `sum_tf/2`, as well as `len/1` and `len_tf/1`.)

Independently of stack allocation, there can be a significant difference in delay when using one fold instead of the other. Take for example the following two calls computing the same value:

`foldl(fun join/2,[],L)` $\equiv$ `foldr(fun join/2,[],L)`.

The first one will be slower than the second. See exercise on page 73.

What cannot be programmed with folds? As the defining properties show, folds traverse the input list in its entirety, hence there is no way to get off the bus while it is running. For instance, `rm_fst/2` on page 80,

```
rm_fst(_,   []) $\xrightarrow{\alpha}$ [];
rm_fst(I,[I|L]) $\xrightarrow{\beta}$ L;
rm_fst(I,[J|L]) $\xrightarrow{\gamma}$ [J|rm_fst(I,L)].
```

cannot be implemented by means of a fold because there are two ends to the function calls: either the item has not been found and we ran afoul the end of the list in clause α, or it has been found somewhere inside the list in clause β. However, in theory, if we accept a full traversal of the list for every call, then `rm_fst/2` can be programmed by means of a rightward fold. The usual technique is to have an accumulator which is either an atom meaning “not found” or a pair with an atom meaning “found” and the current built list. If the result is “not found” then we just return the original list. The following function makes a stronger case because it is really impossible to express by means of a fold, even inefficiently. It is the “complete tail” function:

```
ctail(   []) -> [];
ctail([_|L]) -> L.
```

In general, a function F can be equivalently expressed by a call to a fold if, and only if, for all lists P and Q, for all item I, we have

$$F\texttt{(}P\texttt{)} \equiv F\texttt{(}Q\texttt{)} \Rightarrow F\texttt{([}I\texttt{|}P\texttt{])} \equiv F\texttt{([}I\texttt{|}Q\texttt{])}. \tag{12.19}$$

We have `ctail([])` $\equiv$ `ctail([2])` but `ctail([1])` $\not\equiv$ `ctail([1,2])`.

One positive side-effect of using maps and folds is that they sometimes allow the programmer to recognise some compositions that can be optimised by means of some algebraic transformation of the abstract syntax tree. As an example of a simple equation, we have, for all functions F and G:

$$\texttt{map(}F\texttt{,map(}G\texttt{,}L\texttt{))} \equiv \texttt{map(compose(}F\texttt{,}G\texttt{),}L\texttt{)}.$$

Without counting in the delays of F and G, the left-hand side induces the delay $2n+2$, if L contains n items, whereas the right-hand side incurs the delay $n+2$, so it is much preferable to the former. Another interesting equation is

$$\texttt{foldl(}F\texttt{,}A\texttt{,}L\texttt{)} \equiv \texttt{foldr(}F\texttt{,}A\texttt{,}L\texttt{)} \tag{12.20}$$

if F is associative and symmetric. Let us prove it. The first clause of the definition of `foldl/3` and the first clause of the definition of `foldr/3` imply that, for all F and A,

$$\texttt{foldl(}F\texttt{,}A\texttt{,[])} \equiv A \equiv \texttt{foldr(}F\texttt{,}A\texttt{,[])}.$$

For non-empty lists, this equation means:

$$F(I_{n-1}, \ldots, F(I_1, F(I_0, A)) \ldots) \equiv F(I_0, F(I_1, \ldots, F(I_{n-1}, A)) \ldots).$$

Although the ellipses in the previous equation are intuitive, they are not a valid foundation for a rigorous mathematical argument. By definition, we have

$$\texttt{foldl(}F\texttt{,}A\texttt{,[}I\texttt{|}L\texttt{])} \equiv \texttt{foldl(}F\texttt{,}F\texttt{(}I\texttt{,}A\texttt{),}L\texttt{)}.$$

Dually, by definition, we also have

$$\texttt{foldr(}F\texttt{,}A\texttt{,[}I\texttt{|}L\texttt{])} \equiv F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,}L\texttt{))}.$$

The original equation would thus be established for all lists if we prove

$$\texttt{foldl(}F\texttt{,}F\texttt{(}I\texttt{,}A\texttt{),}L\texttt{)} \equiv F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,}L\texttt{))}.$$

Let us call this conjecture $\mathcal{P}$ and prove it by *structural induction*. This principle states that, given a finite data structure S, a property $\mathcal{P}$ to be proved about it, then

1. if $\mathcal{P}$ is provable for all the atomic S, that is, configurations of S that cannot be decomposed;

2. if, assuming that $\mathcal{P}$ is proved for all immediate substructures of S, then $\mathcal{P}(S)$ is proved;
3. then $\mathcal{P}(S)$ is proved for *all* S.

The data structure S being a list here, actually named L, there is a unique atomic list: the empty list. So we must first prove $\mathcal{P}(\texttt{[]})$. The first clause of the definition of `foldl/3` and the first clause of the definition of `foldr/3` imply that, for all F and A,

$$\texttt{foldl(}F\texttt{,}F\texttt{(}I\texttt{,}A\texttt{),[])} \equiv F\texttt{(}I\texttt{,}A\texttt{)} \equiv F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,[]))},$$

which is $\mathcal{P}(\texttt{[]})$. Next, let us consider a non-empty list $\texttt{[}J\texttt{|}L\texttt{]}$. What are its immediate substructures? By construction of lists, there is only one immediate sub-list of $\texttt{[}I\texttt{|}L\texttt{]}$, namely L. Therefore, let us assume $\mathcal{P}(L)$ for a given list L and suppose that F is associative and symmetric (This is called the *structural induction hypothesis*) and let us prove $\mathcal{P}(\texttt{[}J\texttt{|}L\texttt{]})$ for all J. For F to be associative means that, for all values X, Y and Z, we have

$$F(X, F(Y, Z)) \equiv F(F(X, Y), Z).$$

The symmetry of F means that, for all X and Y, we have

$$F(X, Y) \equiv F(Y, X).$$

Let us start with the left-hand side of $\mathcal{P}(\texttt{[}J\texttt{|}L\texttt{]})$:

$$\begin{aligned}
&\texttt{foldl(}F\texttt{,}F\texttt{(}I\texttt{,}A\texttt{),[}J\texttt{|}L\texttt{])}\\
&\quad\equiv \texttt{foldl(}F\texttt{,}F\texttt{(}J\texttt{,}F\texttt{(}I\texttt{,}A\texttt{)),}L\texttt{)}, && \text{by definition of } \texttt{foldl/3},\\
&\quad\equiv \texttt{foldl(}F\texttt{,}F\texttt{(}F\texttt{(}J\texttt{,}I\texttt{),}A\texttt{),}L\texttt{)}, && \text{by associativity of } F,\\
&\quad\equiv \texttt{foldl(}F\texttt{,}F\texttt{(}F\texttt{(}I\texttt{,}J\texttt{),}A\texttt{),}L\texttt{)}, && \text{by symmetry of } F,\\
&\quad\equiv F\texttt{(}F\texttt{(}I\texttt{,}J\texttt{),foldr(}F\texttt{,}A\texttt{,}L\texttt{))} && \text{by induction hypothesis } \mathcal{P}(L),\\
&\quad\equiv F\texttt{(}I\texttt{,}F\texttt{(}J\texttt{,foldr(}F\texttt{,}A\texttt{,}L\texttt{)))} && \text{by associativity of } F,\\
&\quad\equiv F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,[}J\texttt{|}L\texttt{]))}, && \text{by definition of } \texttt{foldr/3}.
\end{aligned}$$

This proves $\mathcal{P}(\texttt{[}J\texttt{|}L\texttt{]})$. The principle of structural induction then implies that $\mathcal{P}(L)$ is proved for all lists L, hence the original equation (12.20) on page 269. The previous derivation suggests a variation in the definition of `foldl/3`:

```
foldl_alt(_,A,   []) -> A;
foldl_alt(F,A,[I|L]) -> foldl_alt(F,F(A,I),L).
```

The difference lies in the order of the parameters of `F`. We would then have to prove a slightly different conjecture:

$$\texttt{foldl\_alt(}F\texttt{,}F\texttt{(}A\texttt{,}I\texttt{),}L\texttt{)} \equiv F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,}L\texttt{))}.$$

The previous derivation would read now as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{foldl\_alt}(F\texttt{,}F\texttt{(}A\texttt{,}I\texttt{),[}J\texttt{|}L\texttt{])}} \\
\equiv \texttt{foldl\_alt}(F\texttt{,}F\texttt{(}F\texttt{(}A\texttt{,}I\texttt{),}J\texttt{),}L\texttt{)}, & \text{by definition,} \\
\equiv \texttt{fold\_alt}(F\texttt{,}F\texttt{(}A\texttt{,}F\texttt{(}I\texttt{,}J\texttt{)),}L\texttt{)} & \text{by associativity of } F, \\
\equiv F\texttt{(}F\texttt{(}I\texttt{,}J\texttt{),foldr(}F\texttt{,}A\texttt{,}L\texttt{))}, & \text{by induction hypothesis } \mathcal{P}(L), \\
\equiv F\texttt{(}I\texttt{,}F\texttt{(}J\texttt{,foldr(}F\texttt{,}A\texttt{,}L\texttt{)))} & \text{by associativity of } F, \\
\equiv F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,[}J\texttt{|}L\texttt{]))}, & \text{by definition of } \texttt{foldr/3}.
\end{array}
$$

We see that the symmetry of F is not required anymore but in only one remaining place: when the list is empty. Indeed, we have

$$
\begin{array}{rl}
\texttt{foldl\_alt}(F\texttt{,}F\texttt{(}A\texttt{,}I\texttt{),[])} \equiv F\texttt{(}A\texttt{,}I\texttt{)}, & \text{by definition.} \\
F\texttt{(}I\texttt{,foldr(}F\texttt{,}A\texttt{,[]))} \equiv F\texttt{(}I\texttt{,}A\texttt{)}, & \text{by definition of } \texttt{foldr/3}.
\end{array}
$$

Therefore, in order to prove the variant conjecture about `foldl_alt/3` and `foldr/3`, we must have

$$F\texttt{(}A\texttt{,}I\texttt{)} \equiv F\texttt{(}I\texttt{,}A\texttt{)},$$

that is, A, which is the initial value of the accumulator, must be symmetric with all items I through F. This is not a clear improvement over the first theorem about `foldl/3`, which required all pairs of successive items to be symmetric. Nevertheless, there is an interesting special case, which is when A is a neutral element for F, that is, for all I,

$$F\texttt{(}A\texttt{,}I\texttt{)} \equiv F\texttt{(}I\texttt{,}A\texttt{)} \equiv A.$$

Then symmetry altogether is no more required. Therefore, `foldl_alt/3` is preferable over `foldl/3`, because it provides more opportunities when transforming applications of `foldr/3`. But, since the standard library of Erlang offers the definition `foldl/3`, we shall stick to it. The standard library of OCaml, however, proposes the function `fold_left`, which corresponds to `foldl_alt/3`.

Anyway, theorem (12.20) allows us to transform immediately some calls to `foldr/3`, which requires an amount of control stack at least proportional to the size of the input list, into calls to `foldl/3`, whose parameter F is the only function possibly not using a small and constant amount of control stack (if it does, the gain is immediate and obvious). This is why the following definitions are equivalent:

```
lenl(L) -> foldl(fun(_,A) -> A+1 end,0,L).
lenr(L) -> foldr(fun(_,A) -> A+1 end,0,L).
```

Proving that the following two definitions are equivalent happens to be a bit trickier:

```
suml([N|L]) -> foldl(fun(I,A) -> I+A end,N,L).
```

```
sumr([N|L]) -> foldr(fun(I,A) -> I+A end,N,L).
```

The reason is that the first item of the list serves as the initial value of the accumulator in both cases, despite the order of traversal of the list being reversed (rightward versus leftward). Actually, we already proved that they were equivalent on page 29. It is much more obvious to see that the following definitions are equivalent:

```
sum_tf(L=[_|_]) -> foldl(fun(I,A) -> I+A end,0,L).
sum(L=[_|_])    -> foldr(fun(I,A) -> I+A end,0,L).
```

only because addition is associative and symmetric.

Mappings without association lists. In order to illustrate further the expressive power of higher-order functions, let us muse about a small, albeit unlikely, example. We mentioned on page 260 association lists being a collection of pairs key-value, straightforwardly implemented with lists, for instance, `[{a,0},{b,1},{a,5}]`. A *mapping* is an association list which is searched based on the first component of the pairs. Typically, we have

```
find(_,       []) -> absent;
find(I,[{I,V}|L]) -> V;                 % Associated value found
find(I,    [_|L]) -> find(I,L).                 % Keep searching
```

Notice that if a key is repeated, only the first pair will be considered, for instance, `find(a,[{a,0},{b,1},{a,5}])` evaluates to `0`, not `5`. These pairs are called *bindings*. Let us assume that we want to present formally what a mapping is but without relying upon any particular programming language. In this case, we must count on mathematics to convey the concept, more precisely, on mathematical functions. We would say that a mapping M is a function from some finite set of values $\mathcal{K}$ to some finite set of values $\mathcal{V}$. Therefore, what was previously the conjunction of a data type (a list) and a lookup function (`find/2`) is now a single function, representing the mapping *and* the lookup at the same time. A binding $x \mapsto y$ is just another notation for the pair (x, y), where $x \in \mathcal{K}$ and $y \in \mathcal{V}$. We need now to express how a mapping is updated, that is, how a mapping is extended with new bindings. With a list, this is simply done by pushing a new pair but, without a list, we would say that an update is a function itself, taking a mapping and a binding as arguments and returning a new mapping. *An update is thus a higher-order function.* Let the function $(\oplus)$ be such that $M \oplus x_1 \mapsto y$ is the *update* of the mapping M by the binding $x_1 \mapsto y$, as defined by

$$(M \oplus x_1 \mapsto y)(x_2) := \begin{cases} y & \text{if } x_1 = x_2, \\ M(x_2) & \text{otherwise.} \end{cases}$$

We can check that we return the value associated to the first key matching the input, as expected. The empty mapping would be a special function returning a special symbol meaning "not found," like $M_\varnothing(x) = \perp$, for all x. The mapping containing the binding $(1,5)$ would be $M_\varnothing \oplus 1 \mapsto 5$. This is very abstract and independent of any programming language, whilst being totally precise. If now the need arises to show how this definition can be programmed, this is when functional languages can shine. An update would be directly written in Erlang as

```
update(M,{X1,Y}) -> fun(X2) -> case X2 of X1 -> Y;
                                          _  -> M(X2)
                               end
                    end.
```

The correspondence with the formal definition is almost immediate, there is no need to introduce a data structure and its interpretation. The empty mapping is simply

```
empty(_) -> absent.
```

For example, the mapping as list `[{a,0},{b,1},{a,5}]` can be modelled with higher-order functions only as

```
    update(update(update(fun empty/1,{a,5}),{b,1}),{a,0}).
```

Perhaps what needs to be learnt from all this is that lists in functional languages, despite having a distinctive syntax and being used pervasively, are not a fundamental data type: functions are.

Functional encoding of tuples. Let us start by abstracting the tuple into its essence and, because in a functional language, functions are the main feature, we should ask ourselves what is *done* with something we think of as a tuple. Actually, we rushed because we should have realised first that all tuples can actually be expressed in terms of the empty tuple and *pairs*, that is, tuples with exactly two components. For instance, `{5,foo,{fun(X) -> X*X end}}` can be rewritten as `{5,{foo,{fun(X) -> X*X end,{}}}}`. So let us rephrase the question in terms of pairs only. Basically, a pair is constructed (or *injected*) and matched, that is, deconstructed (or *projected*). In the latter, two components can result: either the first component of the pair or the second. This analysis leads to the conclusion that the functional encoding of pairs requires three functions: one for making, `mk_pair/2`, and two for unmaking, `fst/1` and `snd/1`. Once a pair is built, it is represented as a function, therefore functions extracting the components take as an argument another function denoting the pair, thus they are higher-order functions. Consider

```
mk_pair(X,Y) $\xrightarrow{\alpha}$ fun(Pr) $\xrightarrow{\beta}$ Pr(X,Y) end.     %Pr is a projection
fst(P) $\xrightarrow{\gamma}$ P(fun(X,_) $\xrightarrow{\delta}$ X end).            % P denotes a pair
snd(P) $\xrightarrow{\epsilon}$ P(fun(_,Y) $\xrightarrow{\zeta}$ Y end).
```

We have the following expected behaviour:

```
fst(mk_pair(3,5)) $\xrightarrow{\alpha}$ fst(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5))
                  $\xrightarrow{\gamma}$ (fun(Pr) $\xrightarrow{\beta}$ Pr(3,5))(fun(X,_) $\xrightarrow{\delta}$ X end)
                  $\xrightarrow{\beta}$ (fun(X,_) $\xrightarrow{\delta}$ X end)(3,5)
                  $\xrightarrow{\delta}$ 3.
```

To proof the versatility of this encoding, let us define a function `add/1` which adds the components of the pair passed to it:

```
add(P) $\xrightarrow{\eta}$ fst(P) + snd(P).
```

A call to `add/1` would unravel as follows, assuming that arguments are evaluated rightward:

```
add(mk_pair(3,5))
        $\xrightarrow{\alpha}$ add(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)
        $\xrightarrow{\eta}$   fst(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)
          + snd(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)
        $\xrightarrow{\gamma}$   (fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)(fun(X,_) $\xrightarrow{\delta}$ X end)
          + snd(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)
        $\xrightarrow{\beta}$   (fun(X,_) $\xrightarrow{\delta}$ X end)(3,5)
          + snd(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)
        $\xrightarrow{\delta}$ 3 + snd(fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)
        $\xrightarrow{\epsilon}$ 3 + (fun(Pr) $\xrightarrow{\beta}$ Pr(3,5) end)(fun(_,Y) $\xrightarrow{\zeta}$ Y end)
        $\xrightarrow{\beta}$ 3 + (fun(_,Y) $\xrightarrow{\zeta}$ Y end)(3,5)
        $\xrightarrow{\zeta}$ 3 + 5 = 8.
```

This is very nicely done but the keen reader may feel cheated, though, because we could have simply defined `add/2` as

```
add(X,Y) -> X + Y.
```

Indeed, this critique is valid. The ability of functions to receive various arguments at once amounts to them receiving *one* tuple exactly, whose components are these various values. Therefore, we have to retry and make sure that our functions are *nullary* or *unary*, that is, take zero or one argument. This is achieved by taking the one value as argument and rewrite the call into a function which will take in turn the next value as argument etc. This translation is called *currying*.

```
mk_pair(X) $\xrightarrow{\alpha}$ fun(Y) $\xrightarrow{\beta}$ fun(Pr) $\xrightarrow{\gamma}$ (Pr(X))(Y) end end.
fst(P) $\xrightarrow{\delta}$ P(fun(X) $\xrightarrow{\epsilon}$ fun(_) $\xrightarrow{\zeta}$ X end end).
snd(P) $\xrightarrow{\eta}$ P(fun(_) $\xrightarrow{\theta}$ fun(Y) $\xrightarrow{\iota}$ Y end end).
add(P) $\xrightarrow{\kappa}$ fst(P) + snd(P).
```

Remember that writing `fun(X) -> fun(P) -> ...` is the same as `fun(X) -> (fun(P) -> ...)` Also, the parentheses around `Pr(X)` are necessary in Erlang because this call is in the place of a function being called itself. Now we can run the example again and the call now unfolds as follows, supposing that arguments are computed rightward:

```
add((mk_pair(3))(5))
  $\xrightarrow{\alpha}$ add((fun(Y) $\xrightarrow{\beta}$ fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(Y) end end)(5))
  $\xrightarrow{\beta}$ add(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
  $\xrightarrow{\kappa}$   fst(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
     + snd(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
  $\xrightarrow{\delta}$   (fun(Pr)$\xrightarrow{\gamma}$(Pr(3))(5) end)(fun(X)$\xrightarrow{\epsilon}$fun(_)$\xrightarrow{\zeta}$X end end)
     + snd(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
  $\xrightarrow{\gamma}$   ((fun(X) $\xrightarrow{\epsilon}$ fun(_) $\xrightarrow{\zeta}$ X end end)(3))(5)
     + snd(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
  $\xrightarrow{\epsilon}$ (fun(_) $\xrightarrow{\zeta}$ 3 end)(5) + snd(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
  $\xrightarrow{\zeta}$ 3 + snd(fun(Pr) $\xrightarrow{\gamma}$ (Pr(3))(5) end)
  $\xrightarrow{\eta}$ 3
     + (fun(Pr)$\xrightarrow{\gamma}$(Pr(3))(5) end)(fun(_)$\xrightarrow{\theta}$fun(Y)$\xrightarrow{\iota}$Y end end)
  $\xrightarrow{\gamma}$ 3 + ((fun(_) $\xrightarrow{\theta}$ fun(Y) $\xrightarrow{\iota}$ Y end end)(3))(5)
  $\xrightarrow{\theta}$ 3 + (fun(Y) $\xrightarrow{\iota}$ Y end)(5)
  $\xrightarrow{\iota}$ 3 + 5 = 8.
```

Of course, this encoding is usually not worth doing because the number of function calls is much greater than using a data structure.

Functional encoding of lists. To gain more insight into the nature of lists as a data structures, we can encode lists only with higher-order functions. In the former view, a list is an infrastructure, a kind of inert container for data and functions are expected to operate on it. In the latter, it is a composition of functions containing data as arguments and waiting to be called to *do* something with them. The difference between both points of view is not an imagined dichotomy between data and functions, which is blurred in object-oriented languages too, but the fact that higher-order functions *alone* can make up a list.

Since we already know how to encode pairs with higher-order functions, a first approach to encoding lists with functions simply consists in encoding them with pairs. Abstractly, a list can either be empty or constructed by pushing an item into another list, so all we need is to translate these two concepts. The empty list can readily be represented by the empty tuple `{}` and pushing becomes pairing:

```
push(I,L) -> {I,L}.
```

This encoding was introduced on page 151 to save memory on linear accumulators. Here, we want to go one step further and get rid of the pairs themselves by means of their functional interpretation seen above, so `push/2` becomes a renaming of `mk_pair/2`:

```
push(I,L) -> fun(Pr) -> Pr(I,L) end.            % See mk_pair/2
```

To understand the status of the empty list, we must switch to consider projections on lists. These are usually called *head* and *tail*, as we saw long ago on page 19. We implement them as the original versions of `fst/2` and `snd/2`, where `L`, `H` and `T` denote, respectively, an encoding of a list, a head and a tail:

```
head(L) -> L(fun(H,_) -> H end).                       % See fst/2
tail(L) -> L(fun(_,T) -> T end).                       % See snd/2
```

Let us now think *how* the empty list is used. It is a list such that any projection of its putative contents fails, that is, projecting the first component (the head) fails, as well as projecting the second component (the tail). A trick consists in defining

```
empty() -> fail.                       % The atom fail is arbitrary
```

The point is that `empty/0` is nullary, so calling it with an argument fails, as in `head(fun empty/0)`. For example, the list `[a,b,c]` is encoded as

```
push(a,push(b,push(c,fun empty/0))).
```

This solution relies on the *arity*, that is, the number of parameters, of `empty/0` to lead to failure. This failure is consistent with the way classic lists, that is, lists as data structures, are used: `tail([_|L]) -> L` and the call `tail([])` fails to match a clause. The limit of this encoding is that, being based on functions, the encoded lists cannot be matched by the heads of the clauses. For example, the function `ctail/1` on page 268

```
ctail(   []) -> [];
ctail([_|L]) -> L.
```

cannot not be encoded because we would need a way to check whether an encoded list is empty without crashing the program if it is not. If we prefer the caller to be gently informed of the problem instead, that is, we want the definition of the projections to be complete, we could allow `empty/1` to take a projection which is then discarded:

```
empty(_) -> fail.
```

We would have the rewrite `head(fun empty/1)` $\rightarrow$ `fail`, which is not a failure insofar the run-time system is concerned, but is interpreted by the application as a *logical* failure. Of course, it becomes the burden of the caller to check whether the atom `fail` is returned and the burden of the list maker to make sure not to push this atom in the encoded list, otherwise a caller would confuse the empty list with a regular item.

Local recursion. Many functions need some other auxiliary functions to carry out subtasks. For example, consider

```
len_tf(L)    -> len(L,0).
len(   [],N) -> N;
len([_|L],N) -> len(L,N+1).
```

where `len/2` is the auxiliary function, supposed not to be used by any other function. To forbid its usage outside the scope of the module, it is simply omitted in the `-export` clause at the beginning. But it still could be called from within the module it is defined. How could we avoid this as well? This is where anonymous functions comes handy:

```
len_tf(L) ->
  Len = fun(   [],N) -> N;
           ([_|L],N) -> Len(L,N+1)            % Does not compile
        end,
  Len(L,0).
```

This limits the visibility of the anonymous function bound to variable `Len` to the body of `len_tf/1`, which is exactly what we wanted. The problem here is that this definition is rejected by the Erlang compiler because the binding construct (`=`) does not make the variable on its left-hand side visible to the right-side, hence `Len` is unknown in the call `Len(L,N+1)`. In some other functional languages, there is a specific construct to allow recursion on local definitions, for instance, `let rec` in OCaml, but the following hypotyposis is nevertheless theoretically relevant. The original problem becomes another one: how can we define anonymous *recursive* functions? A workaround is to pass an additional function parameter, which is used in stead of the recursive call:

```
len1(L) -> Len = fun(_,   [],N) -> N;
                    (F,[_|L],N) -> F(F,L,N+1)
                 end,
           Len(Len,L,0).
```

Notice that we renamed `len_tf/1` into `len1/1` because we are going to envisage several variants. Moreover, the anonymous function is not equivalent to `len/2` because it takes three arguments. Also, the compiler emits the following warning (we removed the line number):

```
Warning: variable 'L' shadowed in 'fun'.
```

We have seen this before, on page 264. Here, the shadowing is harmless, because inside the anonymous function denoted by `Len`, the original value of `L`, that is, the argument of `len1/1`, is not needed. Nevertheless, for the sake of tranquillity, a simple renaming will get rid of the warning:

```
len1(L) -> Len = fun(_,   [],N) -> N;
                    (F,[_|P],N) -> F(F,P,N+1)          % Renaming
                 end,
           Len(Len,L,0).
```

We can alter this definition by currying the anonymous function and renaming it so `Len` now is equivalent to `fun len/2`:

```
len2(L) -> H = fun(F) -> fun(   [],N) -> N;
                            ([_|P],N) -> (F(F))(P,N+1)
                         end
               end,
           Len = H(H),                    % Equivalent to fun len/2
           Len(L,0).
```

Let us define a function `u/1` which auto-applies its functional argument and let us make use of it in stead of `F(F)`:

```
u(F) -> fun(X,Y) -> (F(F))(X,Y) end.          % Self-application

len3(L) -> H = fun(F) -> fun(   [],N) -> N;
                            ([_|P],N) -> (u(F))(P,N+1)
                         end
               end,
           (H(H))(L,0).                                % Expanded Len
```

Let us replace now `u(F)` by `F`. This transformation does not preserve the semantics of `H`, so let us rename the resulting function `G` and we redefine `H` to be equivalent to its prior instance:

```
len3(L) -> G = fun(F) -> fun(   [],N) -> N;
                            ([_|P],N) -> F(P,N+1)
                         end
               end,
           H = fun(F) -> G(u(F)) end,
           (H(H))(L,0).
```

The interesting point is that the anonymous function referred to by variable `G` is very similar to `Len` at the beginning. (It may sound paradoxical to speak of anonymous functions with names, but, in Erlang, variables and function names are two distinct syntactic categories, so there is no contradiction in terms.) Here it is again:

```
len_tf(L) ->
  Len = fun(   [],N) -> N;
```

```
          ([_|L],N) -> Len(L,N+1)       % Unfortunately invalid
        end,
  Len(L,0).
```

The difference is that `G` abstracts over `F` instead of having a (problematic) recursive call. Let us expand back the call `u(F)` and getting rid of `u/1` altogether:

```
len4(L) ->
  G = fun(F) -> fun(   [],N) -> N;
                   ([_|P],N) -> F(P,N+1)
                end
      end,
  H = fun(F) -> G(fun(X,Y) -> (F(F))(X,Y) end) end,
  (H(H))(L,0).
```

To gain some generality, we can extract the assignments to `H` and `Len`, put them into a new function `y/1` and expand `Len` in place:

```
y(G) -> H=fun(F) -> G(fun(X,Y)->(F(F))(X,Y) end) end, H(H).

len5(L) -> G = fun(F) -> fun(   [],N) -> N;
                            ([_|P],N) -> F(P,N+1)
                         end
               end,
           (y(G))(L,0).
```

By putting the definition of function `y/1` into a dedicated module, we can now easily define recursive anonymous functions. There is a limitation, though, which is that `y/1` is tied to the arity of `F`. For instance, we cannot use it for the factorial:

```
fact(N) -> G = fun(F) -> fun(0) -> 1;
                            (N) -> N * F(N-1)
                         end
               end,
           (y(G))(L,0).                         % Arity mismatch
```

Therefore, if we really want a general scheme, we should work with fully curried functions, so all functions are unary:

```
y(G) -> H = fun(F) -> G(fun(X) -> (F(F))(X) end) end, H(H).

len6(L) -> G=fun(F) -> fun(N) -> fun(   []) -> N;
                                    ([_|P]) -> (F(N+1))(P)
                                 end
                       end
```

```
            end,
          ((y2(G))(0))(L).
```

Notice that we swapped the order of the list and the integer, since there is no pattern matching to be done on the latter. The grammar of Erlang obliges us to put parentheses around every function call resulting in a function being immediately called, so calling fully curried functions with all their arguments, like `((y2(G))(0))(L)`, ends up being a bit fastidious, although a good text editor can help us in paring properly the parentheses.

The theoretical point of this derivation is that we can write recursive functions without relying on recursion at all, since even `y/1` is not recursive. In fact, nothing special is required as long as we have function calls unrestricted by typing rules. Some strongly and statically typed languages like OCaml reject the definition of `y/1` above precisely because of this, but other valid, albeit more complex, definitions are possible. (In the case of OCaml, the switch `-rectypes` allows us to compile the one above, though.) If we grant ourselves the use of recursion, which we never banned, we can actually write a simpler definition of `y/1`, named `y__/1`:

```
y__(F) -> fun(X) -> (F(y__(F)))(X) end.              % Recursive
```

This definition is actually very easy to come by, as it relies on the computational equivalence, for all `X`,

$$\texttt{(y\_\_(F))(X)} \equiv \texttt{(F(y\_\_(F)))(X)},$$

If we assume the mathematical property $\forall x.f(x) = g(x) \Rightarrow f = g$, the previous equivalence would yield

$$\texttt{y\_\_(F)} \equiv \texttt{F(y\_\_(F))},$$

which, by definition, shows that `y__(F)` is a fixpoint of `F`. Beware that

```
y__(F) -> F(y__(F)).                                % Infinite loop
```

does not work because the call `y__(`F`)` would *immediately* start evaluating the call `y__(`F`)` in the body, therefore never ending. Some functional programming languages have a different rewrite strategy than Erlang and do not always start by evaluating the argument in a function call, making this definition directly workable. Another example:

```
fact(N) -> F = fun(F) -> fun(A) -> fun(0) -> A;
                                      (M) -> (F(A*M))(M-1)
                                   end
                         end
             end,
          ((y__(F))(1))(N).
```

The technique we developed in the previous lines can be used to reduce the amount of control stack in some functions. For example, consider

```
join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

Note how the parameter `Q` is threaded until the first argument is empty. This means that a reference to the original `Q` is duplicated at each rewrite until the last step, because the definition is not in tail form. In order to avoid this, we could use a recursive anonymous function which captures `Q` not as a parameter but as part of the (embedding) scope:

```
join(P,Q) -> G = fun(F) -> fun(   []) -> Q;        % Q in scope
                              ([I|R]) -> [I|F(R)]
                           end
                 end,
             (y__(G))(P).
```

This transformation is called *lambda-dropping* and its inverse *lambda-lifting*. The function `y__/1` is called the *Y fixpoint combinator*.

We sometimes may want to define two mutually recursive anonymous functions. Consider the following example, which is in practice utterly useless and inefficient, but simple enough to illustrate our point.

```
even(0) -> true;
even(N) -> odd(N-1).

odd(0) -> false;
odd(N) -> even(N-1).
```

Let us say that we do not want `even/1` to be callable from any other function but `odd/1`. This means that we want the following pattern:

```
odd(I) -> Even = fun(▭) -> ▭ end,
          Odd  = fun(▭) -> ▭ end,
          Odd(I).
```

where `Even` and `Odd` depend on each other. As the canvas is laid out, `Even` cannot call `Odd` and `Odd` can only call `Even`. The technique to allow mutual recursion consists in abstracting the first function over the second, that is, `Even` becomes a function whose parameter is a function destined to be used as `Odd`:

```
odd(I) -> Even = fun(Odd) -> fun(0) -> true;
                                (N) -> Odd(N-1)
                             end
                 end,
          Odd  = fun(▭) -> ▭ end,
          Odd(I).
```

This breaks the (lexical) dependence of `Even` on `Odd`. The next step is more tricky. We can start naively, though, and let the problem come to the fore by itself:

```
odd(I) -> Even = fun(Odd) -> fun(0) -> true;
                                (N) -> Odd(N-1)
                             end
                 end,
          Odd  = fun(0) -> false;
                    (N) -> (Even( Odd ))(N-1)
                 end,
          Odd(I).
```

The problem is not unheard of and we already know how to define an anonymous recursive function by abstracting over the recursive call and passing the resulting function to a fixpoint combinator:

```
odd(I) -> Even = fun(Odd) -> fun(0) -> true;
                                (N) -> Odd(N-1)
                             end
                 end,
          Odd  = y(fun(F) -> fun(0) -> false;
                                (N) -> (Even(F))(N-1)
                             end
                   end),
          Odd(I).
```

Exercises. [See answers page 384.]

1. Write a definition for the function `max_f/3` such that the call `max_f(`F`,`A`,`B`)` is the maximum value of the integer $F(X)$ when integer X ranges between integers A and B, with $A \leqslant B$.
2. Write a definition for the function `max_x/3` such that the call `max_-x(`F`,`A`,`B`)` is the smallest integer X ranging between integers A and B, with $A \leqslant B$, such that the integer $F(X)$ is maximum.

Chapter 13

Merge Sort

Merging. Let us recall one-way insertion sort, as seen on page 209:

```
insert(I,[J|S]) when I > J -> [J|insert(I,S)];
insert(I,    S)            -> [I|S].
isort(   [])               -> [];
isort([I|L])               -> insert(I,isort(L)).
```

The purpose of `insert/2` is to insert one item into a list of items ordered nondecreasingly. Sorting by insertion consists in inserting items to be sorted one by one into an already sorted list. A generalisation is concerned with inserting a series of items *already sorted* in nondecreasing order. This basic operation is called *merging* and it is more efficient than inserting items one at a time, because we don't need to move backwards to insert. Consider the following example, where two increasingly sorted lists, `[22,47,50]` and `[16,36,59,70]` are merged step by step into one increasingly sorted list which appears at the right. The numbers being compared are the heads of the lists and they are set in a bold typeface:

$$\left.\begin{array}{l}\texttt{[}\mathbf{22}\texttt{,47,50]}\\ \texttt{[}\mathbf{16}\texttt{,36,59,70]}\end{array}\right\}\texttt{[]}\quad\text{then}\quad\left.\begin{array}{l}\texttt{[}\mathbf{22}\texttt{,47,50]}\\ \texttt{[}\mathbf{36}\texttt{,59,70]}\end{array}\right\}\texttt{[16]}$$

$$\left.\begin{array}{l}\texttt{[}\mathbf{47}\texttt{,50]}\\ \texttt{[}\mathbf{36}\texttt{,59,70]}\end{array}\right\}\texttt{[16,22]}\quad\text{then}\quad\left.\begin{array}{l}\texttt{[}\mathbf{47}\texttt{,50]}\\ \texttt{[}\mathbf{59}\texttt{,70]}\end{array}\right\}\texttt{[16,22,36]}$$

$$\left.\begin{array}{l}\texttt{[}\mathbf{50}\texttt{]}\\ \texttt{[}\mathbf{59}\texttt{,70]}\end{array}\right\}\texttt{[16,22,36,47]}\quad\text{then}\quad\left.\begin{array}{l}\texttt{[]}\\ \texttt{[59,70]}\end{array}\right\}\texttt{[16,22,36,47,50]}.$$

When a list is empty, we can append it to the current result and obtain `[16,22,36,47,50,59,70]`. Otherwise, we compare the heads of both lists, select the smallest as being the head of the result from now on and recur to the same process without that number. Here is the Erlang code to implement this algorithm:

```
merge(   [],      Q)            $\xrightarrow{\alpha}$ Q;
merge(    P,     [])            $\xrightarrow{\beta}$ P;
merge([I|P],Q=[J|_]) when I < J $\xrightarrow{\gamma}$ [I|merge(P,Q)];
merge(    P,  [J|Q])            $\xrightarrow{\delta}$ [J|merge(P,Q)].
```

Notice that we do not actually need to join lists, as the control context of the recursive calls suffice for obtaining the desired order. The difference with `insert/2` above is that we don’t stop after the first item is inserted, but keep inserting forward until one list or the other is empty, because the remaining list is already sorted. Let us illustrate this method on a short example:

```
merge([3,4,7],[1,2,5,6]) $\xrightarrow{\delta}$ [1|merge([3,4,7],[2,5,6])]
                         $\xrightarrow{\delta}$ [1,2|merge([3,4,7],[5,6])]
                         $\xrightarrow{\gamma}$ [1,2,3|merge([4,7],[5,6])]
                         $\xrightarrow{\gamma}$ [1,2,3,4|merge([7],[5,6])]
                         $\xrightarrow{\delta}$ [1,2,3,4,5|merge([7],[6])]
                         $\xrightarrow{\delta}$ [1,2,3,4,5,6|merge([7],[])]
                         $\xrightarrow{\beta}$ [1,2,3,4,5,6,7].
```

As can be surmised now, merging can efficiently be used as a basic operation for sorting, instead of insertion. Before we proceed to see how, note that we could have defined `merge/2` as follows:

```
merge(    P,     [])            -> P;
merge([I|P],Q=[J|_]) when I < J -> [I|merge(P,Q)];
merge(    P,  [J|Q])            -> [J|merge(P,Q)].
```

This is the previous definition without its first clause α. This shorter version is slower when the first list is shorter than the other: the second list will have all its items examined, although it is unnecessary since they are the only remaining ones. It is faster to stop as soon as the first list becomes empty. We recall that, on page 44, we defined `join/2` as

```
join(   [],Q) -> Q;
join([E|P],Q) -> [E|join(P,Q)].
```

instead of

```
join(    P,[]) -> P;
join(   [], Q) -> Q;
join([E|P], Q) -> [E|join(P,Q)].
```

even if the later is faster when the second list is empty from the beginning. The reason was that this simplification, despite slowing down the program in this special case, made the expression of the delay dependent only upon the length of the first list. This avoided making two cases and allowed us to simply define $\mathcal{D}_n^{\mathtt{join}}$ instead of $\mathcal{D}_{n,m}^{\mathtt{join}}$. The difference with `merge/2` is that `join/2` is not symmetric in general, that is, `join(`P`,`Q`)` $\not\equiv$ `join(`Q`,`P`)`, when both P and Q are not empty, whilst `merge(`P`,`Q`)` $\equiv$ `merge(`Q`,`P`)` always holds. The symmetry of `merge/2` means that we should not arbitrarily distinguish one list with respect to the other, so the delay of `merge(`P`,`Q`)` is unconditionally the same as the delay of `merge(`Q`,`P`)`, that is, $\mathcal{D}_{m,n}^{\mathtt{merge}} = \mathcal{D}_{n,m}^{\mathtt{merge}}$. This results in easier delay calculations and faster computations.

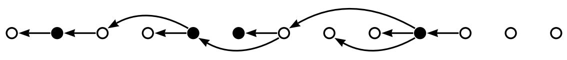


FIGURE 65: Two ordered lists merged into one

Best delay. Let $\mathcal{B}_{m,n}^{\mathtt{merge}}$ be the delay in the best case of `merge(`P`,`Q`)`, where P and Q contain respectively m and n items. The goal here is to minimise the use of clauses γ and δ, that is, the number of comparisons. Graphically, we represent an item from P as a *white node* ($\circ$) and an item from Q as a *black node* ($\bullet$). Nodes of these kinds are printed in an horizontal line, the leftmost node being the smallest. Comparisons are always performed between black and white nodes and are represented as *edges* in FIGURE 65 on page 285. An incoming arrow means that the node is smaller than the other end of the edge, so all edges point leftward and the number of comparisons is the number of nodes with an incoming edge. This abstract representation suggests that the more items from one list we have at the end of the result, the less comparisons we needed for merging (there are two consecutive white nodes at the right without any edges). Indeed, *the minimum delay is achieved when the shortest list comes first in the result, that is, it is a* prefix. See FIGURE 66 on page 285 for the best case with $m = 9$ and $n = 4$, which are the same lengths as in FIGURE 65 on page 285. If both lists have the same length, any of them can be considered the shortest and the previous statement applies as well.

$$\mathcal{B}_{m,n}^{\mathtt{merge}} = 1 + \min\{m, n\}. \tag{13.21}$$

We can check that $\mathcal{B}_{m,n}^{\mathtt{merge}} = \mathcal{B}_{n,m}^{\mathtt{merge}}$ and $\mathcal{B}_{0,n}^{\mathtt{merge}} = \mathcal{B}_{m,0}^{\mathtt{merge}} = 1$, as expected.

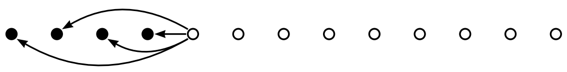


FIGURE 66: Best-case merging: $\min\{m, n\}$ comparisons

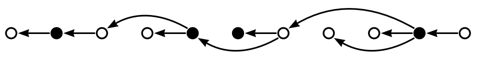


FIGURE 67: A worst-case merging in $m + n - 1$ comparisons

Worst delay. Let $\mathcal{W}^{\mathsf{merge}}_{m,n}$ be the delay of `merge(`P`,`Q`)` in the worst case, where P and Q contain respectively m and n items. Clearly, this delay is one, by clause α or β, plus the number of times clauses γ and δ are used, which is the number of comparisons needed. For the purpose of illustration, let us consider an example where $m = 9$ and $n = 4$. A possible merging of P and Q is shown in FIGURE 65 on page 285. We can see that we can increase the number of comparisons with respect to $m+n$ by removing those nodes from Q which find their place at the end of the result *without being compared*. This is shown in FIGURE 67 on page 286. After this shortening, we have $m + n = 11$, that is, down by 2, and the number of comparisons is unchanged ($m+n-1 = 10$). This is the maximum number because it corresponds to a situation where all the nodes, except the rightmost, are the destination of an edge. (The last node cannot be pointed at because edges must go from right to left.) Actually, it is possible to interchange the last two nodes as well, because this removes one out-going edge from the white node but adds one to the black, so the number of comparisons is still $m + n - 1$. This can be visualised in FIGURE 68 on the following page. *The worst case of merging P and Q is when the last item of one list becomes the last item of the result and the last item of the other list becomes the penultimate item of the result.* The total number of comparisons is then $m + n - 1$, where m and n are, respectively, the number of items in P and Q. When one list is empty, the delay is 1. Therefore, when $m > 0$ and $n > 0$,

$$\mathcal{W}^{\mathsf{merge}}_{0,n} = \mathcal{W}^{\mathsf{merge}}_{m,0} = 1; \quad \mathcal{W}^{\mathsf{merge}}_{m,n} = 1 + (m + n - 1) = m + n. \tag{13.22}$$

It is important to realise that the above reckoning is unchanged if we swap P and Q, thus m and n; formally, we expect $\mathcal{W}^{\mathsf{merge}}_{m,n} = \mathcal{W}^{\mathsf{merge}}_{n,m}$.

Average delay. One question remains to be answered: Is the average delay closer to the lower or the upper bound of the delay? Let us consider a smaller example in FIGURE 69 on the next page, with $m = 3$ white nodes and $n = 2$ black nodes which are interleaved in all possible

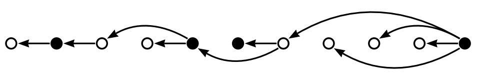


FIGURE 68: The two rightmost nodes in FIGURE 67 on the following page have been swapped

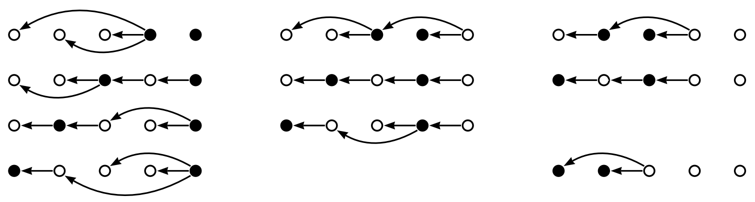


FIGURE 69: All possible merges with $m = 3$ (○) and $n = 2$ (●)

manners. Note how the figure was structured. The first column lists the configurations where the rightmost black node is the last of the result. The second column lists the cases where the rightmost black node is the penultimate node of the result. The third column is divided in two groups itself, the first of which lists the cases where the rightmost black node is the antepenultimate. The total number of comparisons is 35 and the number of configurations is 10, thus the average number of comparisons is $35/10 = 7/2$. Let us devise a method to find this ratio for any m and n. First, the number of configurations: how many ways are there to combine m white nodes and n black nodes? This is the same as asking how many ways there are to paint in black n nodes picked amongst $m+n$ white nodes. More abstractly, this is equivalent to wonder how many ways there are to choose n objects amongst $m+n$. This number is usually a *combination* and noted $\binom{m+n}{n}$. For example, let us consider the set $\{a, b, c, d, e\}$ and the combinations of 3 objects taken from it are

$$\begin{gathered}\{a,b,c\}, \{a,b,d\}, \{a,b,e\}, \{a,c,d\}, \{a,c,e\}, \{a,d,e\},\\ \{b,c,d\}, \{b,c,e\}, \{b,d,e\},\\ \{c,d,e\}.\end{gathered}$$

This enumeration establishes that $\binom{5}{3} = 10$. (Notice that we use mathematical sets, therefore the order of the elements or their repetition are not meaningful.) It is not difficult to count the combinations if we recall how we counted the permutations, on page 195. Let us determine $\binom{r}{k}$. We can pick the first object amongst r, the second amongst $r - 1$ etc. until we pick the rth object amongst $r - k + 1$, so there are $r(r - 1) \ldots (r - k + 1)$ choices. But these arrangements contain duplicates, for example, we may form $\{a, b, c\}$ and $\{b, a, c\}$, which are to be considered identical combinations. Therefore, we must divide the number we just obtained by the number of redundant arrangements, which is the number of permutations of k objects, $k!$. In the end:

$$\binom{r}{k} := \frac{r(r-1)\dots(r-k+1)}{k!} = \frac{r!}{k!(r-k)!}.$$

We can check now that in FIGURE 69 on the next page, we must have 10 cases: $\binom{5}{2} = 5!/(2!3!) = 10$. The symmetry of the problem means that merging a list of m items with a list of n leads to exactly the same results as merging a list of n items with a list of m items.

$$\binom{m+n}{n} = \binom{m+n}{m}.$$

This can be easily proved by means of the definition:

$$\binom{m+n}{n} := \frac{(m+n)!}{n!(m+n-n)!} = \frac{(m+n)!}{m!n!} := \binom{m+n}{m}, \quad \text{QED}.$$

Going forth, let us determine the total number $\mathcal{L}(m,n)$ of comparisons needed to merge m and n items in all possible manners. This is the number of times clauses γ and δ are used, so we have to count the nodes which are the destination of an edge in our graph representation. In *enumerative combinatorics*, that is, the art of counting configurations, it is often worthy to consider the complementary set. In this case, it may be easier to count the nodes without incoming edges, that is, the number of times clauses α and β are used. These nodes are circled, so they are easily distinguished, in FIGURE 70 on page 288. We can check for each merge that their number is always $m+n$ minus the number of nodes with incoming edges. This is easy to justify because there are $m+n$ nodes and each is either processed by clauses α-β or else clauses γ-δ. Summing over all the $\binom{m+n}{n}$ merges, we draw that the total number of circled nodes is

$$(m+n)\binom{m+n}{n} - \mathcal{L}(m,n).$$

It is simple to characterise these circled nodes: they make up the longest rightmost contiguous series of nodes of the same colour. Since there are

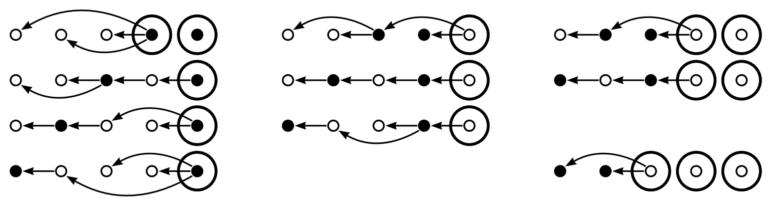


FIGURE 70: All merges with nodes handled by clauses α and β circled

only two colours, the problem of determining the total number $W(m,n)$ of white circled nodes is symmetric to the determination of the total number $B(m,n)$ of black circled nodes: $W(m,n) = B(n,m)$. Bear in mind that, in our running example, m is the number of white nodes and n the number of black nodes. We expect then

$$(m+n)\binom{m+n}{n} - \mathcal{L}(m,n) = W(m,n) + B(m,n),$$

hence, dividing by $\binom{m+n}{n}$ and reordering the terms yields

$$\Big[\mathcal{L}(m,n)\Big] \Big/ \binom{m+n}{n} = m + n - \Big[W(m,n) + B(m,n)\Big] \Big/ \binom{m+n}{n}.$$

The left-hand side is the average number of comparisons we are looking for in the end. We can decompose $W(m,n)$ by counting the circled white nodes vertically, which is another common technique used in combinatorics. In FIGURE 70 on page 288, $W(3,2)$ is the sum of the numbers of merges with

- one ending circled white node (6),
- two ending circled white nodes (3),
- three ending circled white nodes (1).

Indeed, the examination of the two last columns in FIGURE 70 on page 288 brings $W(3,2) = 6 + 3 + 1 = 10$ and the first column $B(3,2) = 4 + 1 = 5$. How can we generalise to arbitrary m and n? The number of merges with one ending circled white node is the answer to the question: "How many ways are there to combine n black nodes with $m-1$ white nodes?" We know, by definition of combinations, this to be $\binom{n+(m-1)}{n}$. Similarly, the number of merges with two ending circled white nodes is the answer to the question: "How many ways are there to combine n black nodes with $m-2$ white nodes?" This is $\binom{n+(m-2)}{n}$. So on until $\binom{n+(m-m)}{n}$:

$$\begin{aligned}
W(m,n) &= \binom{n+m-1}{n} + \binom{n+m-2}{n} + \cdots + \binom{n+0}{n}\\
&= \binom{n+0}{n} + \cdots + \binom{n+m-2}{n} + \binom{n+m-1}{n},\\
W(m,n) &= \sum_{j=0}^{m-1} \binom{n+j}{n}. \qquad (13.23)
\end{aligned}$$

This sum can actually be simplified, more precisely, it has a closed form, but in order to understand its underpinnings, we need firstly to develop our intuition about combinations. By computing combinations

$\binom{r}{k}$		k									
		0	1	2	3	4	5	6	7	8	9
r	0	**1**	0	0	0	0	0	0	0	0	0
	1	**1**	**1**	0	0	0	0	0	0	0	0
	2	**1**	**2**	**1**	0	0	0	0	0	0	0
	3	**1**	**3**	**3**	**1**	0	0	0	0	0	0
	4	**1**	**4**	**6**	**4**	**1**	0	0	0	0	0
	5	**1**	**5**	**10**	**10**	**5**	**1**	0	0	0	0
	6	**1**	**6**	**15**	**20**	**15**	**6**	**1**	0	0	0
	7	**1**	**7**	**21**	**35**	**35**	**21**	**7**	**1**	0	0
	8	**1**	**8**	**28**	**56**	**70**	**56**	**28**	**8**	**1**	0
	9	**1**	**9**	**36**	**84**	**126**	**126**	**84**	**36**	**9**	**1**

FIGURE 71: The corner of Pascal's Triangle (in bold)

$\binom{r}{k}$ for small values of r and k using the definition, we can fill a table traditionally known as *Pascal's triangle* and displayed in FIGURE 71 on page 290. Note how we set the convention $\binom{r}{k} = 0$ if $k > r$. Pascal's Triangle contains many interesting properties relative to the sum of some of its values. For instance, if we choose a number in the triangle and look at the one on its right, then the one below the latter is their sum. For the sake of illustration, let us extract from FIGURE 71 on page 290 the lines $r = 7$ and $r = 9$:

	7	**1**	**7**	**21**	**35**	**35**	**21**	**7**	**1**	0	0
	8	**1**	**8**	**28**	**56**	**70**	**56**	**28**	**8**	**1**	0

We surrounded two examples of the additive property of combinations we discussed: $21 + 35 = 56$ and $21 + 7 = 28$. We would then bet that

$$\binom{r-1}{k-1} + \binom{r-1}{k} = \binom{r}{k}.$$

This is actually not difficult to prove if we go back to the definition:

$$\binom{r}{k} := \frac{r!}{k!(r-k)!} = \frac{r}{k} \cdot \frac{(r-1)!}{(k-1)!((r-1)-(k-1))!} = \frac{r}{k}\binom{r-1}{k-1}.$$

$$\binom{r}{k} := \frac{r!}{k!(r-k)!} = \frac{r}{r-k} \cdot \frac{(r-1)!}{k!((r-1)-k)!} = \frac{r}{r-k}\binom{r-1}{k}.$$

The first equality is valid if $k > 0$ and the second if $r \neq k$. We can now replace $\binom{r-1}{k-1}$ and $\binom{r-1}{k}$ in terms of $\binom{r}{k}$ in the sum

$$\binom{r-1}{k-1} + \binom{r-1}{k} = \frac{k}{r}\binom{r}{k} + \frac{r-k}{r}\binom{r}{k} = \binom{r}{k}, \quad \text{QED}.$$

The sum is valid if $r > 0$. There is direct proof of the formula, which does not involve computations but pure logic. Let us suppose that we *already* have all the subsets of k items chosen among r. By definition, there are $\binom{r}{k}$ of them. We choose to distinguish an arbitrary item amongst r and we want to split the collection in two subsets: on one side, all the combinations containing this particular item, on the other side, all the combinations without it. The former subset has cardinal $\binom{r-1}{k-1}$ because its combinations are built from the fixed item and further completed by choosing $k-1$ remaining items amongst $r-1$. The latter subset is made of $\binom{r-1}{k}$ combinations which are made from $r-1$ items, of which k have to be selected because we ignore the distinguished item. This yields the same additive formula. Now, let us return to our pending sum

$$\sum_{j=0}^{m-1} \binom{n+j}{n}.$$

1	0
5	**1**
15	**6**
35	**21**
	56

In terms of navigation across Pascal's Triangle, we understand that this sum operates on numbers in the same column. More precisely, it starts from the diagonal with the number $\binom{n}{n}$ and goes down until a total of m numbers have been added. So let us choose a simple example and fetch two adjacent columns were the sum is small. On the left is an excerpt for $n = 4$ (the left column is the fifth in Pascal's Triangle) and $m = 4$ (height of the left column). Interestingly, the sum of the numbers in bold in the left column, that is, the sum under study, equals the number in bold at the bottom of the second column: $1 + 5 + 15 + 35 = 56$. By checking other columns, we may feel justified to think that this is a general pattern. Before attempting a general proof, let us see how it may work on our particular example. Let us start from the bottom of the second column, that is, 56, and use the addition formula in reverse, that is, express 56 as the sum of the numbers in the row above it: $56 = 35 + 21$.

35	**21**
	56

Graphically, this sum is seen as the triangle at the left, extracted from the previous one. The number 35 has just appeared and we would like to keep it because it is part of the equality to prove. So let us apply the addition formula again to 21 and draw $21 = 15 + 6$. Let us keep 15 and resume the same procedure on 6 so $6 = 5+1$. Finally, $1 = 1+0$. We just checked $56 = 35+(15+(5+(1+0)))$, which is exactly what we wanted. Because we want the number corresponding to 35 in our example to be $\binom{n+m-1}{n}$, we have the derivation

$$\binom{n+m}{n+1} = \binom{n+m-1}{n} + \binom{n+m-1}{n+1}$$

$$= \binom{n+m-1}{n} + \left[\binom{n+m-2}{n} + \binom{n+m-2}{n+1}\right]$$
$$= \binom{n+m-1}{n} + \binom{n+m-2}{n} + \cdots + \left[\binom{n}{n} + \binom{n}{n+1}\right],$$
$$\binom{n+m}{n+1} = \sum_{j=0}^{m-1}\binom{n+j}{n} = W(m,n), \text{ by eq. (13.23) on page 289.}$$

By symmetry, $B(m,n) = W(n,m)$, therefore $B(m,n) = \binom{m+n}{m+1}$. We can check these formulas for $m = 3$ and $n = 2$: $W(3,2) = \binom{5}{3} = 5!/(3!2!) = 10$ and $B(3,2) = \binom{5}{4} = 5!/(4!1!) = 5$, both being correct. Let $\mathcal{A}^{\gamma\delta}(m,n)$ be the average number of comparisons to merge m and n numbers, which is the same as the number of times clauses γ or δ are used. We found on the next page that

$$\mathcal{A}^{\gamma\delta}(m,n) = m + n - \Big[W(m,n) + B(m,n)\Big] \Big/ \binom{m+n}{n}.$$

We can now replace $W(m,n)$ and $B(m,n)$ and obtain a closed form:

$$\mathcal{A}^{\gamma\delta}(m,n) = m + n - \left[\binom{m+n}{n+1} + \binom{m+n}{m+1}\right] \Big/ \binom{m+n}{n}$$
$$= m + n - \frac{n!m!}{(n+1)!(m-1)!} - \frac{n!m!}{(m+1)!(n-1)!},$$
$$\mathcal{A}^{\gamma\delta}(m,n) = m + n - \frac{m}{n+1} - \frac{n}{m+1} = \frac{mn}{m+1} + \frac{mn}{n+1}.$$

Let us check this pretty formula with $\mathcal{A}^{\gamma\delta}(3,2) = 6/4 + 6/3 = 7/2$, which is the value we computed on page 287. Let us recall the program as found on page 284:

```
merge(   [],      Q)               $\xrightarrow{\alpha}$ Q;
merge(    P,     [])               $\xrightarrow{\beta}$ P;
merge([I|P],Q=[J|_]) when I < J $\xrightarrow{\gamma}$ [I|merge(P,Q)];
merge(    P,  [J|Q])               $\xrightarrow{\delta}$ [J|merge(P,Q)].
```

Because each call to `merge/2` ends either with a rewrite by clause α or β, the relationship between the average delay of `merge/2`, noted $\mathcal{A}^{\mathsf{merge}}_{m,m}$, and $\mathcal{A}^{\gamma\delta}(m,n)$ is simply

$$\mathcal{A}^{\mathsf{merge}}_{m,n} = 1 + \mathcal{A}^{\gamma\delta}(m,n) = \frac{mn}{m+1} + \frac{mn}{n+1} + 1. \tag{13.24}$$

First, let us check that $\mathcal{A}^{\mathsf{merge}}_{m,n} = \mathcal{A}^{\mathsf{merge}}_{n,m}$, which was expected because every symmetric problem has a symmetric solution. Next, we can now verify what happens when $m = 0$ or $n = 0$, as we expect the number of comparisons to be zero and we indeed deduce $\mathcal{A}^{\mathsf{merge}}_{0,n} = \mathcal{A}^{\mathsf{merge}}_{m,0} = 1$. This proves that the average delay in this case coincides with the minimum delay and, consequently, the lower bound in $\mathcal{B}^{\texttt{merge}}_{m,n} \leqslant \mathcal{A}^{\texttt{merge}}_{m,n} \leqslant \mathcal{W}^{\texttt{merge}}_{m,n}$. In fact, if $m > 0$ and $n \geqslant 0$ or $n > 0$ and $m \geqslant 0$, we have

$$1 + \min\{m, n\} \leqslant \mathcal{A}^{\texttt{merge}}_{m,n} \leqslant m + n,$$

and the bounds are actually tight. Another case is worth pondering: $m = 1$ or $n = 1$. Indeed, we may realise then that merging a singleton list with another list leads to the same result as inserting the single item into the other list. Let us recall the `insert/2` function used in insertion sort, on page 209:

```
insert(I,[J|S]) when I > J -β-> [J|insert(I,S)];
insert(I,    S)            -γ-> [I|S].
```

We then have, for all items I and all lists L, the equivalence

$$\texttt{insert(}I\texttt{,}L\texttt{)} \equiv \texttt{merge([}I\texttt{],}L\texttt{)}.$$

Therefore, *insertion is a special case of merging.* Nevertheless, the average delays are not exactly the same:

$$\mathcal{A}^{\texttt{insert}}_{n} = \frac{1}{2}n + 1, \quad \mathcal{A}^{\texttt{merge}}_{1,n} = \frac{1}{2}n + 2 - \frac{1}{n+1},$$

as we found on page 212. Therefore, `merge/2` is slightly slower in average than `insert/2` in this special case:

$$\mathcal{A}^{\texttt{merge}}_{1,n} - \mathcal{A}^{\texttt{insert}}_{n} = 1 - \frac{1}{n+1} < 1.$$

Nonetheless, asymptotically, this difference is insignificant:

$$\mathcal{A}^{\texttt{merge}}_{1,n} \sim \mathcal{A}^{\texttt{insert}}_{n}, \text{ as } n \to \infty.$$

Also, it may be interesting to see what happens when $m = n$, that is, when the two lists to be merged have the same length:

$$\mathcal{A}^{\texttt{merge}}_{n,n} = \frac{2n^2}{n+1} + 1 \sim 2n, \text{ as } n \to \infty. \tag{13.25}$$

In other words, the average delay of merging two lists of identical length is asymptotically the total number of items being merged.

Merging can be used to sort a list of items if this list can be previously split evenly in two sub-lists which are in turn sorted, taking into account that a singleton list is always sorted by definition. This recursive approach to sorting is said *top-down* because of the way it is usually depicted, the original list being written at the top of the page, the singletons being located at the bottom. Another view first maps the list to be sorted into a list of singletons, directly at the bottom. As we noticed before, these can be readily merged pairwise and their results too etc. until one sorted list remains at the top. Here, the sorted

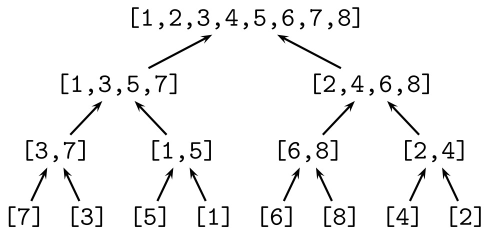


FIGURE 72: Sorting bottom-up `[7,3,5,1,6,8,4,2]` by merging

list is at the top, not the input list. This approach is said *bottom-up* because, after the singletons have been constructed, the merging takes place directly. In the top-down way, these singletons are reached after a succession of splittings. Anyway, the same detail proves thorny: how to manage the cases when, in the top-down fashion, a list to be evenly divided contains an odd number of items or how, in the bottom-up way, an odd number of lists can be merged pairwise.

Bottom-up merge sort of 2^p items. Let us ignore these problems for now and explore a small example with a number of items which is a power of 2, like `[7,3,5,1,6,8,4,2]`. Sorting by merging, be it top-down or bottom-up, results in the same kind of graphical representation: a *complete binary tree*. We introduced trees on page 18 and a binary tree is a tree whose nodes have either no subtree or exactly two, as shown in FIGURE 72 on page 294. This structure mirrors faithfully the strategy: if reading top-down, we divide a problem (to sort a list) into two subproblems (to sort two smaller lists); if reading bottom-up, we reduce two solutions (two small sorted lists) into one solution (a bigger sorted list). FIGURE 72 on page 294 illustrates the bottom-up approach of starting off from singletons and merging them and the resulting lists in turn etc. until the root of the tree is an ordered list. Perhaps the first question arising is how to obtain all these singletons at the leaves of the tree. This is actually quite easy if we already rely on a map, as seen on page 263:

```
map(_,   []) -> [];
map(F,[I|L]) -> [F(I)|map(F,L)].
```

Then, the function `solo/1` reduces simply to one call:

```
solo(L) -> map(fun(I) -> [I] end,L).
```

Otherwise, we opt for the equivalent handmade

```
solo(   []) -> [];
solo([I|L]) -> [[I]|solo(L)].
```

The delay of `solo(L)` when L contains n items is $\mathcal{D}_n^{\texttt{solo}} = n + 1$ with the latter definition. We can now refocus on FIGURE 72 on the following page and try to quantify some of its aspects. For instance, we may wonder about the *height* of the tree or the total number of merges. We assumed that the we sorted 2^p numbers, with $p > 0$. In our example $p = 3$. Since at each step upward in the tree, all the nodes at one level are merged pairwise, there are half the number of parents. There are 2^p leaves, which have 2^{p-1} parents, which have 2^{p-2} parents etc. until $2^0 = 1$, which is the root. This observation allows us to answer our two questions at once. First, it means that there are $p + 1$ levels in the tree, including the root. Second, it implies that there are p merge phases making, bottom-up, 2^{p-1}, 2^{p-2}, ..., 2^0 merges, amounting to $2^{p-1} + 2^{p-2} + \cdots + 2^0$ merges in total. This sum has a simple closed form. Let us consider the general case $Q_x(p) := x^{p-1} + x^{p-2} + \cdots + x^0$. Then $xQ_x(p) - Q_x(p) = x^p - x^0$, that is, when $x \neq 1$,

$$\sum_{k=0}^{p-1} x^k = \frac{x^p - 1}{x - 1}. \tag{13.26}$$

In particular, $x = 2$ let us know that there are $2^p - 1$ merges in total. If, as it is custom, we express the size of the input as $n = 2^p$, we prefer to express these previous expressions in terms of n instead of p. This is easy, since we then have $p = \lg n$, where $\lg x$ is the *binary logarithm* of x, which can be defined as the inverse function of $x \mapsto 2^x$, for $x > 0$. Now we can conclude that the height of the tree is $1 + \lg n$ and the number of merges is $n - 1$. We may also want to quantify the amount of memory needed to sort by merging $n = 2^p$ keys. First, the number of stacks created is the number of nodes of the merge tree: $2^p + 2^{p-1} + \ldots + 2^0 = 2^{p+1} - 1 = 2n - 1$. Because the keys and the empty stack are shared with the input, we should count the references to them. Also, we have to account for the pushes, namely the nodes (`|`): there is one for each reference to a key. This leads us to determine the sum of the lengths of all the created stacks: $(p + 1)2^p = n \lg n + n$. We must add $2n - 1$ for all the references to `[]`, so the total number of references created is $n \lg n + 3n - 1$ and the number of nodes (`|`) is $n \lg n + n$. Finally, we might compute the total delay to sort 2^p items because we know that, bottom-up, the length of the lists doubles and we already studied the delay of merging, in particular when the two lists have the same length, $\mathcal{D}_{n,n}^{\texttt{merge}}$. Consequently, all we need is to resume our calculation of the number of merges whilst making sure to ponder each stage with the corresponding delay of merging. In order to focus on the concepts at stake, we shall leave out the delay $\mathcal{D}_{2^p}^{\texttt{solo}} = 2^p + 1$

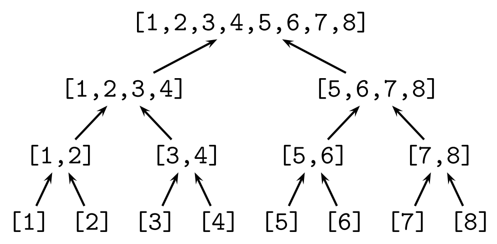


FIGURE 73: A best case for bottom-up merge sort

for now, which can be thought of as preprocessing time. So let us note $\mathcal{D}(n)$ the delay to sort n items without taking into account `solo/1`. We have, for $p > 0$,

$$\begin{aligned}\mathcal{D}(2^p) &= 2^{p-1} \cdot \mathcal{D}^{\text{merge}}_{1,1} + 2^{p-2} \cdot \mathcal{D}^{\text{merge}}_{2,2} + \cdots + 2^0 \cdot \mathcal{D}^{\text{merge}}_{2^{p-1},2^{p-1}} \\ &= \sum_{k=0}^{p-1} \left(2^{p-1-k} \cdot \mathcal{D}^{\text{merge}}_{2^k,2^k}\right) = 2^{p-1} \sum_{k=0}^{p-1} \frac{1}{2^k} \mathcal{D}^{\text{merge}}_{2^k,2^k}. \qquad (13.27)\end{aligned}$$

This sum spurs us to investigate further the best and worst cases of `merge/2` when its arguments have the same length. The result we deduced on page 285 applies here as well, so

$$\mathcal{B}^{\text{merge}}_{n,n} = 1 + \min\{n, n\} = n + 1.$$

This most favourable configuration arises when all the numbers in the shortest list are smaller than the smallest number of the longest. In the present case, both lists have the same length, so it happens whenever all the numbers in one list are smaller than the smallest number in the other. For example, if both lists are singletons, we find ourselves in the best case and the delay is 2. This gives us already $\mathcal{D}^{\text{merge}}_{1,1} = \mathcal{B}^{\text{merge}}_{1,1} = 2$. One possible configuration is when the list is already sorted, as shown in FIGURE 73 on page 296. If the list is sorted in decreasing order, it is also a best case, as can be verified in FIGURE 74 on page 296. Other

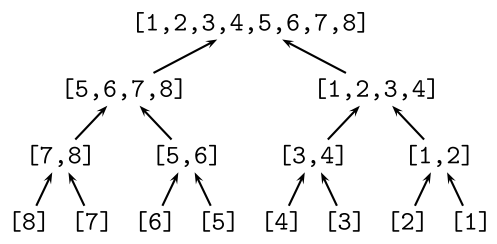


FIGURE 74: Another best case for bottom-up merge sort

variations are possible: in FIGURE 73 on page 296, start from the root and exchange any pair of subtrees which have the same parent and this is still a best case. Importantly, *a best case has the property that it is made of subtrees which are themselves instances of a best case*. This allows us to construct all the best cases. (How many are there when n numbers are given?) Let us note $\mathcal{B}(n)$ the best delay for sorting n items without taking into account the preprocessing by `solo/1`. We can write the delay of bottom-up merge sort in the best case as follows:

$$\mathcal{B}(2^p) = 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}\mathcal{B}^{\texttt{merge}}_{2^k,2^k} = 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}(2^k+1) = 2^{p-1}\left(p + \sum_{k=0}^{p-1}\frac{1}{2^k}\right).$$

We can reuse equation (13.26) on the next page with $x = 1/2$, which implies

$$\sum_{k=0}^{p-1}\frac{1}{2^k} = 2 - \frac{1}{2^{p-1}}. \qquad (13.28)$$

Let us finish now:

$$\mathcal{B}(2^p) = 2^{p-1}\left(p + 2 - \frac{1}{2^{p-1}}\right) = p2^{p-1} + 2^p - 1; \qquad (13.29)$$

$$\mathcal{B}(n) = \frac{1}{2}n\lg n + n - 1, \text{ with } n = 2^p.$$

How does $n\lg n$ compares to the more familiar n^2? Let us prove that, for all real $x > 0$, we have $x < 2^x$. Let us define a function $\xi(x) = 2^x - x$, for $x \geqslant 0$. Its derivative is $\xi'(x) = 2^x - 1$. Because $x \mapsto 2^x$ is a strictly increasing function, $x > 0 \Rightarrow 2^x > 2^0 = 1$. Thus, $\xi'(x) > 0$. Since $\xi(0) = 0$, all this implies $\xi(x) > 0$ for $x > 0$, hence the expected result. Because the logarithm is the inverse of the exponentiation, which is an increasing function, the logarithm is an increasing function. Therefore we can apply it on both sides of an inequality, provided they are positive: $0 < x < 2^x$ implies $\lg x < x$. Furthermore, $n\lg n < n^2$ for all $n > 0$.

Let us delve now into the worst case of sorting bottom-up by merging when $n = 2^p$, for $p > 0$. Equation (13.22) on page 286 was $\mathcal{W}^{\texttt{merge}}_{m,n} = m+n$, thus $\mathcal{W}^{\texttt{merge}}_{n,n} = 2n$. The corresponding configuration supposes that the last number of one of the two lists being merged is the last item of the result and that the last number of the other list finds its place as the penultimate in the result. Obviously, this happens when both lists are singleton, that is, $n = 1$, that is, for the leaves of any tree capturing the essence of bottom-up sorting by merging: $\mathcal{D}^{\texttt{merge}}_{1,1} = \mathcal{W}^{\texttt{merge}}_{1,1} = 2$. It is easy to derive a worst case from the one presented in FIGURE 72 on page 294: just exchange the leaves `[4]` and `[6]`. The result is shown in FIGURE 75 on page 298. Just as with the best case, constructing a

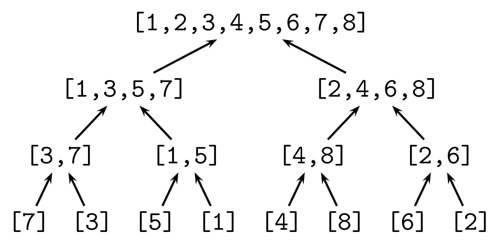


FIGURE 75: A worst case for bottom-up merge sort

global worst case is achieved by constructing worst cases for all the subtrees. Let us note $\mathcal{W}(n)$ the worst delay for sorting n items without counting in the delay due to `solo/1`.

$$\mathcal{W}(2^p) = 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}\mathcal{W}^{\texttt{merge}}_{2^k,2^k} = 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}(2^k+2^k) = p2^p. \quad (13.30)$$

$$\mathcal{W}(n) = n\lg n, \text{ with } n = 2^p.$$

In sum, we have found the following bounds when $n = 2^p$:

$$\mathcal{B}(n) = \tfrac{1}{2}n\lg n + n - 1 \leqslant \mathcal{D}(n) \leqslant n\lg n = \mathcal{W}(n). \quad (13.31)$$

Notice how the lower bound is only a bit more than half the upper bound. Contrast this with insertion sort, where the lower bound was linear in the input size, whilst the upper bound was quadratic. Before jumping to conclusions, bear in mind that we have analysed only a special case of merge sort, when n is a power of 2. We shall deal with the general case latter. But, before doing so, let us make full profit of our analysis of the delay of merging by drawing the average delay of bottom-up merge sort when $n = 2^p$. Equation (13.25) on page 293 was:

$$\mathcal{A}^{\texttt{merge}}_{n,n} = \frac{2n^2}{n+1} + 1 = 2n - 1 + \frac{2}{n+1}.$$

Let us note $\mathcal{A}(n)$ the average delay to sort n items without adding up the delay of the calls to `solo/1`. The average delay, being a delay, obeys the same recurrent equations as the general delay, so it is made of all the average delays of processing the substructures:

$$\begin{aligned}\mathcal{A}(2^p) = 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}\mathcal{A}^{\texttt{merge}}_{2^k,2^k} &= 2^{p-1}\left(2p - \sum_{k=0}^{p-1}\frac{1}{2^k} + 2\sum_{k=0}^{p-1}\frac{1}{2^k(2^k+1)}\right)\\ &= 2^{p-1}\left(2p - \sum_{k=0}^{p-1}\frac{1}{2^k} + 2\sum_{k=0}^{p-1}\left(\frac{1}{2^k} - \frac{1}{2^k+1}\right)\right),\end{aligned}$$

$$\mathcal{A}(2^p) = p2^p + 2^p - 1 - 2^p \sum_{k=0}^{p-1} \frac{1}{2^k + 1}, \text{ with } p > 0. \tag{13.32}$$

There is no closed form known for the remaining sum but, for $k \geqslant 0$, we can work out rough bounds for it and thus for $\mathcal{A}(2^p)$:

$$2^{k+1} = 2^k + 2^k \geqslant 2^k + 1 > 2^k \Rightarrow \frac{1}{2}\sum_{k=0}^{p-1} \frac{1}{2^k} \leqslant \sum_{k=0}^{p-1} \frac{1}{2^k + 1} < \sum_{k=0}^{p-1} \frac{1}{2^k}$$

$$1 - \frac{1}{2^p} \leqslant \sum_{k=0}^{p-1} \frac{1}{2^k + 1} < 2 - \frac{1}{2^{p-1}}, \text{ by eq. (13.28).}$$

Because $1/(2^k+1) > 0$, the partial sums $\sum_{k=0}^{p-1} \frac{1}{2^k+1}$ are positive and increasing in function of p. They are also converging (absolutely) because the limit of the upper bound we found is finite: $\lim_{p\to\infty} (2 - 1/2^{p-1}) = 2$. The limit of the lower bound being 1, it proves that $\sum_{k\geqslant 0} \frac{1}{2^k+1}$ is a number between 1 and 2 and Peter Borwein proved that it is actually an irrational number. Let us call it $\alpha \simeq 1.26449978034844420 92$. Then, we can discard the upper bound $2 - 1/2^{p-1}$ in favour of α:

$$1 - \frac{1}{2^p} \leqslant \sum_{k=0}^{p-1} \frac{1}{2^k + 1} < \alpha \Rightarrow -\alpha 2^p < -2^p \sum_{k=0}^{p-1} \frac{1}{2^k + 1} \leqslant -2^p \left(1 - \frac{1}{2^p}\right).$$

$$p2^p - (\alpha - 1)2^p - 1 < \mathcal{A}(2^p) \leqslant p2^p. \tag{13.33}$$

Finally, setting $n = 2^p$, we conclude

$$n \lg n - (\alpha - 1)n - 1 < \mathcal{A}(n) \leqslant n \lg n. \tag{13.34}$$

This means that the average delay cannot be as small as the best delay, whilst it can be as large as the worst delay. In fact, inequalities (13.34) allow us at last to derive the asymptotic approximation of $\mathcal{A}(n)$:

$$\mathcal{A}(n) \sim n \lg n \sim \mathcal{W}(n), \text{ as } n = 2^p \text{ and } p \to \infty.$$

Therefore, *the average delay is asymptotically equivalent to the worst case delay when sorting* 2^p *items*. It can be proved that the worst delay of `merge/2` is the worst delay of any *optimal* merge algorithm when both lists have the same length (the maximum delay is minimum in this case) and all merges are done on lists of equal lengths when $n = 2^p$. Therefore, the worst delay $\mathcal{W}(n)$ is minimum. After the general case of merge sort has been presented, we will also show that the worst delay of merge sort is asymptotically as good as the worst delay of any optimal sorting algorithm. Another interesting fact we can already underline is that the worst case of merge sort is not sensitive to a possible lack of randomness of the input data. For instance, we saw that the worst delay of insertion sort happens when the data is already sorted. We also

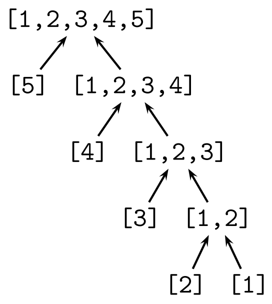


FIGURE 76: Mergings equivalent to the worst case of insertion sort

mentioned before that insertion was a special case of merging, which may lead to a quadratic delay. Therefore, it is crucial for merge sort not to degenerate into insertion sort, by making sure that the maximum number of merges are performed on lists of almost the same length. See the problem in FIGURE 76 on page 300, where the delay is quadratic and, thus, is not an instance of merge sort. This observation will be our guideline in the design of the general bottom-up merge sort.

General bottom-up merge sort. In general, any number can be uniquely expressed as the sum of distinct powers of 2, for example, $7 = 2^2 + 2^1 + 2^0$, and this is the foundation of the *binary number system*. Let us assume $n = \sum_{i=0}^{m-1} b_i 2^i$, where the $b_i \in \{0, 1\}$ are called *bits*. Then $n = 2(b_{m-1}2^{m-2} + \cdots + b_1) + b_0$, which means that b_0 is the remainder of the Euclidean division of n by 2, traditionally noted $b_0 = n \mod 2$. This uniquely defines b_0 because a number is either odd or even. The same division process is repeated on the quotient $\lfloor n/2 \rfloor$, establishing the remaining bits. In binary, 7 is expressed as the series of bits $(111)_2$.

How can we manage the general case of merge sort, that is, when the number of items is not a power of 2? FIGURE 77 on page 300 shows

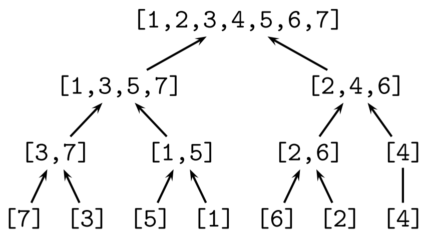


FIGURE 77: Sorting seven numbers bottom-up by merging

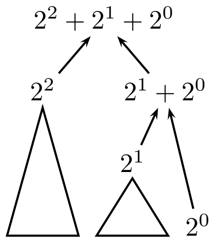


FIGURE 78: Abstraction of FIGURE 77 on page 300

an example where seven numbers are sorted. Notice how the rightmost leaf, that is, `[4]`, is connected by an egde without an arrow to a node above containing `[4]` as well because it is not merged. The list `[4]` is only merged at the next level with `[2,6]`, a list twice its length. The imbalance in length is propagated upward, as `[1,3,5,7]` is merged with `[2,4,6]`. Let us rely on the special case $n = 2^p$ to find an upper bound on $\mathcal{B}(n)$ for arbitrary n, tight enough to yield an asymptotic equivalence with the help of the lower bound.

The general situation can be guessed by abstracting away unnecessary details from the example of FIGURE 77 on page 300. We can see that there are three special subcases: the subtree rooted at `[1,3,5,7]`, figured by a triangle and containing 2^2 numbers at its leaves; the subtree rooted at `[2,6]`, also depicted by a triangle and containing 2^1 leaves, and the subtree whose root is `[4]`, containing 2^0 leaf. See FIGURE 78 on the next page, where only edges corresponding to merges, that is, ending with an arrow, are represented. In all generality, let $n = 2^{e_r} + \cdots + 2^{e_1} + 2^{e_0} > 0$, with $e_r > \cdots > e_1 > e_0 \geqslant 0$ and $r \geqslant 0$. The natural numbers e_i are the positions of the 1-bits in the binary representation of n. In particular, this means that $i \leqslant e_i$ and $r+1$ is the number of 1-bits or, equivalently, the sum of the bits of n. Also, 2^{e_r} being the largest power in the decomposition of n, it is identical to the largest power in the binary expansion of n, so $e_r + 1$ is equal to the number of bits of n. Let m be this number. Then the smallest integer expressible with m bits is $(10\ldots0)_2 = 2^{m-1} = 2^{e_r}$. Dually, the greatest integer is $(1\ldots1)_2 = 2^{m-1} + \cdots + 2^1 + 2^0 = 2^m - 1$. For $n > 0$,

$$2^{m-1} \leqslant n \leqslant 2^m - 1 < 2^m \Rightarrow m - 1 \leqslant \lg n < m \Rightarrow e_r = \lfloor \lg n \rfloor. \quad (13.35)$$

Now let us consider in FIGURE 79 on page 302 the tree of all the merges when we abstract away the details. The triangles are trees made of *balanced merges*, that is, merges performed on lists of the same length, for which we already found the delays. The nodes from the root $2^{e_r} + \cdots + 2^{e_0}$ down to $2^{e_1} + 2^{e_0}$ are the lengths of the results of *unbalanced* *merges*. The total delay is the sum of the balanced and unbalanced merges:

$$\mathcal{D}(n) = \sum_{i=0}^{r} \mathcal{D}(2^{e_i}) + \sum_{i=1}^{r} \mathcal{D}^{\mathsf{merge}}_{2^{e_i}, 2^{e_{i-1}}+\dots+2^{e_0}}. \tag{13.36}$$

If $r = 0$, we found again the special case $n = 2^{e_0}$, as the second sum is zero. Let us bet that the general case is asymptotically the same as the special case $n = 2^p$ and strive for upper and lower bounds on the best, worst and average delays which converge to the same functions. This hope should not been considered as obvious, but really optimistic. For instance, even though $2^{\lfloor \lg n \rfloor} = n$, when n is a power of 2, it is *false* to state that $2^{\lfloor \lg n \rfloor} \sim n$, as $n \to \infty$. Consider the function $x(p) = 2^p - 1$ ranging over the positive integers. First, let us notice that, for all $p > 0$,

$$2^{p-1} \leqslant 2^p - 1 < 2^p \Rightarrow p - 1 \leqslant \lg(2^p - 1) < p \Rightarrow \lfloor \lg(2^p - 1) \rfloor = p - 1.$$

Therefore, $2^{\lfloor \lg(x(p)) \rfloor} = 2^{p-1} = (x(p) + 1)/2 \sim x(p)/2 \nsim x(p)$, which proves that $2^{\lfloor \lg(n) \rfloor} \nsim n$ when $n = 2^p - 1 \to \infty$.

Best delay. The best case of `merge/2` is given by equation (13.21) on page 285, that is, $\mathcal{B}^{\mathsf{merge}}_{m,n} = 1 + \min\{m, n\}$, so

$$\mathcal{B}(n) = \sum_{i=0}^{r} \mathcal{B}(2^{e_i}) + \sum_{i=1}^{r} (1 + \min\{2^{e_i}, 2^{e_{i-1}} + \dots + 2^{e_0}\}).$$

Let us commence by noting that the following inequality holds:

$$\sum_{j=0}^{i} 2^{e_j} \leqslant \sum_{j=0}^{e_i} 2^j = 2 \cdot 2^{e_i} - 1.$$

This is the same as saying that a given binary number is always lower or equal than the number with the same number of bits all set to 1, for example, $(10110111)_2 \leqslant (11111111)_2$. By definition of e_i: $e_{i-1} + 1 \leqslant e_i$,

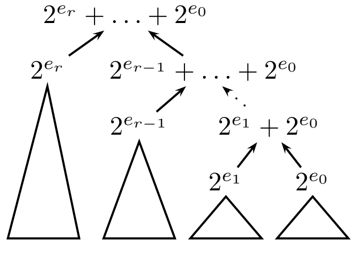


FIGURE 79: Sorting by merging $n = 2^{e_r} + \dots + 2^{e_0}$ items

hence

$$\sum_{j=0}^{i-1} 2^{e_j} \leqslant 2^{e_{i-1}+1} - 1 \leqslant 2^{e_i} - 1 < 2^{e_i}, \tag{13.37}$$

therefore, $\min\{2^{e_i}, 2^{e_{i-1}} + \cdots + 2^{e_0}\} = 2^{e_{i-1}} + \cdots + 2^{e_0}$. We can reformulate the original definition of the best delay now as

$$\mathcal{B}(n) = \sum_{i=0}^{r} \mathcal{B}(2^{e_i}) + r + \sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j}. \tag{13.38}$$

Equation (13.29) on page 297 states $\mathcal{B}(2^p) = p2^{p-1} + 2^p - 1$, hence

$$\sum_{i=0}^{r} \mathcal{B}(2^{e_i}) = \frac{1}{2}\sum_{i=0}^{r} e_i 2^{e_i} + n - (r+1),$$

$$\mathcal{B}(n) = \frac{1}{2}\sum_{i=0}^{r} e_i 2^{e_i} + n - 1 + \sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j}. \tag{13.39}$$

By reusing inequations (13.37), let us work out a small upper bound:

$$\begin{aligned}\sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} &\leqslant \sum_{i=1}^{r}\left(2^{e_{i-1}+1} - 1\right) = 2\sum_{i=0}^{r-1} 2^{e_i} - r \\ &= 2(n - 2^{e_r}) - r = 2n - 2^{\lfloor \lg n\rfloor+1} - \nu_n + 1.\end{aligned} \tag{13.40}$$

The function $\nu_n := r+1$ is the number of 1-bits in the binary expansion of n, or, equivalently, the sum of its bits. This function of n is called *population count*, *sideways sum*, *bit sum* or *Hamming weight*. It can be defined recursively as follows:

$$\nu_0 := 0, \quad \nu_{2n} := \nu_n, \quad \nu_{2n+1} := \nu_n + 1. \tag{13.41}$$

We have the following intuitive tight bounds for any $n > 0$:

$$1 \leqslant \nu_n \leqslant \lfloor \lg n\rfloor + 1,$$

because deduction (13.35) on the current page establishes that $\lfloor \lg n\rfloor + 1$ is the number of bits of n. Notice that ν is a deceptively simple function: first, it is periodic because $\nu_{2^p} = 1$ and, second, $\nu_{2^p-1} = p$. We can check that $n = 2^{e_0}$ leads to both sides to equal 0, as expected. The remaining sum $\sum_{i=0}^{r} e_i 2^{e_i}$ can be simply upper-bounded as follows:

$$\sum_{i=0}^{r} e_i 2^{e_i} \leqslant \sum_{i=0}^{r} e_r 2^{e_i} = e_r \sum_{i=0}^{r} 2^{e_i} = n\lfloor \lg n\rfloor. \tag{13.42}$$

This bound is tight only if $n = 2^{e_0}$. We can conclude now with the following bound, tight when n is a power of 2:

$$\mathcal{B}(n) \leqslant \tfrac{1}{2}n\lfloor \lg n\rfloor + 3n - 2^{\lfloor \lg n\rfloor+1} - \nu_n \leqslant \tfrac{1}{2}n\lfloor \lg n\rfloor + 3n - 2^{\lfloor \lg n\rfloor+1} - 1.$$

In order to ease finding the limit of the bound when n gets large, we need to get rid of the integer parts and the bit sum. We have

$$x - 1 < \lfloor x \rfloor \leqslant x \Rightarrow 2^x < 2^{\lfloor x \rfloor + 1} \leqslant 2^{x+1} \Rightarrow n + 1 \leqslant 2^{\lfloor \lg n \rfloor + 1} \leqslant 2n. \tag{13.43}$$

The lower bound is tight only when $n = 2^p - 1$ and the upper bound when $n = 2^p$. In other words, the expression $2^{\lfloor \lg n \rfloor + 1}$ increases the most between consecutive integers of shape $2^p - 1$ and 2^p. Now we can weaken the upper bound of $\mathcal{B}(n)$, so it becomes simpler:

$$\mathcal{B}(n) \leqslant \tfrac{1}{2} n \lg n + 3n - (n+1) - 1 = \tfrac{1}{2} n \lg n + 2n - 2. \tag{13.44}$$

The bound is tight if and only if $n = 1$ because it assumed that n is both of the form 2^p and $2^p - 1$: this is worse than it could be but we are not looking for the smallest upper bound here, only one which helps us to derive the asymptotic behaviour. Let us endeavour now for a lower bound whose leading term is, hopefully, asymptotically equivalent to $\frac{1}{2} n \lg n$. Let us resume from equation (13.39) on page 303 and look for a lower bound of the first summation $\sum_{i=0}^{r} e_i 2^{e_i}$. Up to this point, we have expressed an arbitrary number $n > 0$ in two ways: first, as a sum of distinct powers of 2 and we wrote $n := \sum_{i=0}^{r} 2^{e_i}$, where $e_r > \ldots > e_1 > e_0 \geqslant 0$ and $r \geqslant 0$ (this point of view focuses on the 1-bits of the binary representation of n); second, as a binary string $n := \sum_{i=0}^{m-1} b_i 2^i = (b_{m-1} \ldots b_0)_2$, where $b_i \in \{0, 1\}$ and $b_{m-1} = 1$. The sequence $(e_i)_i$ is an increasing subsequence of the positive integers, so

$$\sum_{i=0}^{r} e_i 2^{e_i} = \sum_{i=0}^{m-1} i b_i 2^i.$$

It is easy to derive a formula for b_i. Let $n = (b_{m-1} \ldots b_1 b_0)_2$. We have

$$\frac{n}{2^{i+1}} = \frac{1}{2^{i+1}} \sum_{k=0}^{m-1} b_k 2^k = \frac{1}{2^{i+1}} \sum_{k=0}^{i} b_k 2^k + (b_{m-1} \ldots b_{i+1})_2.$$

We want to prove that $\lfloor n/2^{i+1} \rfloor = (b_{m-1} \ldots b_{i+1})_2$, that is,

$$\frac{1}{2^{i+1}} \sum_{k=0}^{i} b_k 2^k < 1 \;\Leftrightarrow\; \sum_{k=0}^{i} b_k 2^k < 2^{i+1} \;\Leftarrow\; \sum_{k=0}^{i} 2^k = 2^{i+1} - 1 < 2^{i+1},$$

which obviously holds. So $2\lfloor n/2^{i+1} \rfloor = (b_{m-1} \ldots b_{i+1} 0)_2$. Similarly, $\lfloor n/2^i \rfloor = (b_{m-1} \ldots b_{i+1} b_i)_2$, so, in general, we have

$$b_i = \left\lfloor \frac{n}{2^i} \right\rfloor - 2 \left\lfloor \frac{n}{2^{i+1}} \right\rfloor.$$

We can now use this latter equation to expand the former:

$$\begin{aligned}\sum_{i=0}^{r} e_i 2^{e_i} &= \sum_{i=0}^{m-1} i\left(\left\lfloor\frac{n}{2^i}\right\rfloor - 2\left\lfloor\frac{n}{2^{i+1}}\right\rfloor\right)2^i\\ &= \sum_{i=0}^{m-1} i\left\lfloor\frac{n}{2^i}\right\rfloor 2^i - \sum_{i=1}^{m}(i-1)\left\lfloor\frac{n}{2^i}\right\rfloor 2^i\\ &= -m\left\lfloor\frac{n}{2^m}\right\rfloor 2^m + \sum_{i=1}^{m}\left\lfloor\frac{n}{2^i}\right\rfloor 2^i = \sum_{i=1}^{m-1}\left\lfloor\frac{n}{2^i}\right\rfloor 2^i,\end{aligned}$$

because $\lfloor n/2^m\rfloor = 0$. Moreover, $\lfloor x\rfloor > x-1$ implies $\lfloor n/2^i\rfloor 2^i > n - 2^i$, so we can proceed as follows:

$$\begin{aligned}\sum_{i=0}^{r} e_i 2^{e_i} &= \sum_{i=1}^{m-1}\left\lfloor\frac{n}{2^i}\right\rfloor 2^i \geqslant \sum_{i=1}^{m-1}(n-2^i) = n(m-1) - \sum_{i=0}^{m-1} 2^i + 2^0\\ &= n(m-1) - (2^m - 1) + 1 = n\lfloor\lg n\rfloor - 2^{\lfloor\lg n\rfloor+1} + 2,\end{aligned} \tag{13.45}$$

by equation (13.35) on the current page: $m - 1 = e_r = \lfloor\lg n\rfloor$. Notice that the bound is tight if and only if $n = 1$ (then $m = 1$ and the two sums are null). This allows us to deduce, from equation (13.39) on the facing page:

$$\mathcal{B}(n) \geqslant (\tfrac{1}{2}n\lfloor\lg n\rfloor - 2^{\lfloor\lg n\rfloor} + 1) + n - 1 + \sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j}.$$

We can easily obtain a lower bound of the remaining double summation by remarking that $e_j \geqslant j$, therefore

$$\begin{aligned}\sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} &= \sum_{j=0}^{r-1} 2^{e_j} + \sum_{i=1}^{r-1}\sum_{j=0}^{i-1} 2^{e_j} = (n - 2^{e_r}) + \sum_{i=1}^{r-1}\sum_{j=0}^{i-1} 2^{e_j}\\ &\geqslant n - 2^{\lfloor\lg n\rfloor} + \sum_{i=1}^{r-1}\sum_{j=0}^{i-1} 2^j = n - 2^{\lfloor\lg n\rfloor} + \sum_{i=1}^{r-1}(2^i - 1)\\ &= n - 2^{\lfloor\lg n\rfloor} + 2^{\nu_n - 1} - \nu_n.\end{aligned} \tag{13.46}$$

because $e_r = \lfloor\lg n\rfloor$ and $\nu_n := r + 1$. The bound is tight when $r = 0$, that is, $n = 2^{e_0}$. Let us resume:

$$\begin{aligned}\mathcal{B}(n) &\geqslant (\tfrac{1}{2}n\lfloor\lg n\rfloor - 2^{\lfloor\lg n\rfloor} + 1) + n - 1 + (n - 2^{\lfloor\lg n\rfloor} + 2^{\nu_n - 1} - \nu_n)\\ &= \tfrac{1}{2}n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+1} + 2^{\nu_n-1} - \nu_n.\end{aligned}$$

Let $f(x) := 2^{x-1} - x$ a function defined on real numbers x. The derivative is $f'(x) = 2^{x-1}\ln 2 - 1$, which is zero for $x = -\lg(\ln 2) \simeq 1.53$ and positive afterwards. We also easily check that $f(1) = f(2) = 0, f(3) = 1$ and $f(4) = 4$, so we are now certain that $2^{\nu_n-1} - \nu_n \geqslant 0$ for any integer $n > 0$, as shown in FIGURE 80 on page 307.

$$\mathcal{B}(n) \geqslant \tfrac{1}{2}n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+1}. \tag{13.47}$$

We can weaken the bound with the inequalities (13.43) on the following page:

$$\mathcal{B}(n) > \tfrac{1}{2}n(\lg n - 1) + 2n - 2n = \tfrac{1}{2}n\lg n - \tfrac{1}{2}n.$$

Together with inequation (13.44) on the current page, we finally have proved

$$\tfrac{1}{2}n\lg n - \tfrac{1}{2}n < \mathcal{B}(n) \leqslant \tfrac{1}{2}n\lg n + 2n - 2.$$

Now we can draw the asymptotic approximation

$$\mathcal{B}(n) \sim \tfrac{1}{2}n\lg n, \text{ as } n \to \infty.$$

As a reminder, the case $n = 2^{e_0}$ in equation (13.29) on page 297 was

$$\mathcal{B}(n) = \tfrac{1}{2}n\lg n + n - 1.$$

Worst delay. Let us turn our attention now to the worst delay. We resume the mathematical definition at equation (13.36) on page 302:

$$\mathcal{D}(n) = \sum_{i=0}^{r}\mathcal{D}(2^{e_i}) + \sum_{i=1}^{r}\mathcal{D}^{\texttt{merge}}_{2^{e_i},2^{e_{i-1}}+\cdots+2^{e_0}}.$$

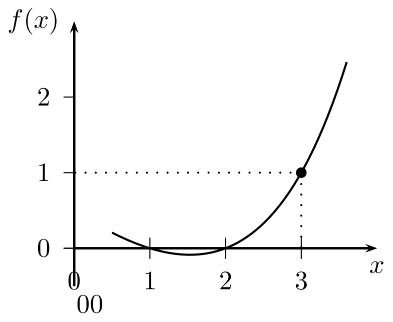
The worst delay of `merge/2` is given by equation (13.22) on page 286, that is, $\mathcal{W}^{\texttt{merge}}_{m,n} = m + n$, with $m > 0$ and $n > 0$, therefore, using equation (13.30) on page 297, that is, $\mathcal{W}(2^p) = p2^p$, we draw

$$\mathcal{W}(n) = \sum_{i=0}^{r}\mathcal{W}(2^{e_i}) + \sum_{i=1}^{r}\sum_{j=0}^{i}2^{e_j} = \sum_{i=0}^{r}e_i2^{e_i} + \sum_{i=1}^{r}\sum_{j=0}^{i}2^{e_j}. \tag{13.48}$$

If $r = 0$, this equation degenerates into the correct $\mathcal{W}(n) = \mathcal{W}(2^{e_0})$ because the double sum cancels out completely. Let us focus on it:

$$\begin{aligned}\sum_{i=1}^{r}\sum_{j=0}^{i}2^{e_j} = \sum_{i=1}^{r}\left(2^{e_i} + \sum_{j=0}^{i-1}2^{e_j}\right) &= (n - 2^{e_0}) + \sum_{i=1}^{r}\sum_{j=0}^{i-1}2^{e_j}\\ &\geqslant n - 2^{e_0} + (n - 2^{\lfloor\lg n\rfloor} + 2^{\nu_n-1} - \nu_n)\\ &\geqslant 2n - 2^{\lfloor\lg n\rfloor+1} + 2^{\nu_n-1} - \nu_n. \end{aligned} \tag{13.49}$$

We made use of inequality (13.46) on the previous page. We also have $e_0 \leqslant \lfloor\lg n\rfloor$. The bound is tight only when $n = 2^{e_0}$. We can replace this



1-11 001

tempcolor[rgb]blacktempcolor[rgb]white

inequality back into the definition of $\mathcal{W}(n)$:

$$\mathcal{W}(n) \geqslant \sum_{i=0}^{r} e_i 2^{e_i} + 2n - 2^{\lfloor \lg n \rfloor + 1} + 2^{\nu_n - 1} - \nu_n.$$

We already bounded the remaining summation in inequation (13.45) on the preceding page, so we deduce

$$\mathcal{W}(n) \geqslant n\lfloor \lg n \rfloor + 2n - 2^{\lfloor \lg n \rfloor + 2} + 2^{\nu_n - 1} - \nu_n + 2.$$

The bound is tight if $n = 1$. Let us weaken it as we did before in order to obtain a more legible bound. We already know that $2^{\nu_n - 1} - \nu_n \geqslant 0$ for any integer $n > 0$, as shown in FIGURE 80 on the next page, so

$$\mathcal{W}(n) \geqslant n\lfloor \lg n \rfloor + 2n - 2^{\lfloor \lg n \rfloor + 2} + 2. \tag{13.50}$$

We can weaken it further in the usual manner:

$$\mathcal{W}(n) > n(\lg n - 1) + 2n - 2(2n) + 2 = n \lg n - 3n + 2. \tag{13.51}$$

Let us turn now to finding a small upper bound for $\mathcal{W}(n)$. We already know how to bound $\sum_{i=1}^{r} \sum_{j=0}^{i-1} 2^{e_j}$ by means of inequation (13.40) on page 303, hence

$$\sum_{i=1}^{r} \sum_{j=0}^{i} 2^{e_j} \leqslant (n - 2^{e_0}) + (2n - 2^{\lfloor \lg n \rfloor + 1} - \nu_n + 1) \leqslant 3n - 2^{\lfloor \lg n \rfloor + 1} - \nu_n.$$

The remaining sum was bounded in inequation (13.42) on page 303, that is, $\sum_{i=0}^{r} e_i 2^{e_i} \leqslant n\lfloor \lg n \rfloor$, so we can draw an upper bound for $\mathcal{W}(n)$:

$$\mathcal{W}(n) \leqslant n\lfloor \lg n \rfloor + 3n - 2^{\lfloor \lg n \rfloor + 1} - \nu_n.$$

The bound is tight only when $n = 1$. As we did previously, we trade some weakeness for some legibility:

$$\mathcal{W}(n) \leqslant n\lfloor \lg n \rfloor + 3n - 2^{\lfloor \lg n \rfloor + 1} - 1. \tag{13.52}$$

Inequations (13.43) on page 304 lead to $-(n+1) \geqslant -2^{\lfloor \lg n \rfloor + 1}$, so

$$\mathcal{W}(n) \leqslant n \lg n + 3n - (n+1) - 1 = n \lg n + 2n - 2. \tag{13.53}$$

Together with inequation (13.51), we thus proved

$$n \lg n - 3n + 2 < \mathcal{W}(n) \leqslant n \lg n + 2n - 2,$$

which imply the same asymptotic behaviour in the special case $n = 2^{e_0}$:

$$\mathcal{W}(n) \sim n \lg n, \text{ as } n \to \infty.$$

As a reminder, the worst delay when $n = 2^{e_0}$ in (13.30) on page 297 is

$$\mathcal{W}(n) = n \lg n.$$

Average delay. The remaining general case to investigate is the average delay $\mathcal{A}(n)$. We can restart with equation (13.36) on page 302

$$\mathcal{A}(n) = \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + \sum_{i=1}^{r} \mathcal{A}^{\texttt{merge}}_{2^{e_i},2^{e_{i-1}}+\cdots+2^{e_0}}.$$

The average delay of `merge/2` is given by equation (13.24) on page 292:

$$\mathcal{A}^{\texttt{merge}}_{p,q} = \frac{pq}{p+1} + \frac{pq}{q+1} + 1.$$

We draw

$$\mathcal{A}(n) = \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + \sum_{i=1}^{r} \left(\frac{2^{e_i}\sum_{j=0}^{i-1} 2^{e_j}}{2^{e_i}+1} + \frac{2^{e_i}\sum_{j=0}^{i-1} 2^{e_j}}{\sum_{j=0}^{i-1} 2^{e_j}+1} + 1 \right). \tag{13.54}$$

From inequations (13.37) on page 303, $\sum_{j=0}^{i-1} 2^{e_j} \leqslant 2^{e_i} - 1$, we deduce

$$\frac{1}{\sum_{j=0}^{i-1} 2^{e_j}+1} \geqslant \frac{1}{2^{e_i}} \implies \frac{1}{2^{e_i}+1} + \frac{1}{\sum_{j=0}^{i-1} 2^{e_j}+1} \geqslant \frac{2^{e_i+1}+1}{2^{e_i}(2^{e_i}+1)}.$$

The left-hand side is $\mathcal{A}^{\texttt{merge}}_{p,q}/p/q$. Multiplying by pq:

$$2^{e_i}\sum_{j=0}^{i-1} 2^{e_j} \left(\frac{1}{2^{e_i}+1} + \frac{1}{\sum_{j=0}^{i-1} 2^{e_j}+1} \right) \geqslant \frac{2^{e_i+1}+1}{2^{e_i}+1} \sum_{j=0}^{i-1} 2^{e_j}$$
$$= \left(2 - \frac{1}{2^{e_i}+1}\right) \sum_{j=0}^{i-1} 2^{e_j} > \left(2 - \frac{1}{2^{e_i}}\right) \sum_{j=0}^{i-1} 2^{e_j} = 2\sum_{j=0}^{i-1} 2^{e_j} - \sum_{j=0}^{i-1} 2^{e_j-e_i}.$$

In the remaining sums, we have $j < i$, and we know that the integer sequence $(e_i)_i$ is strictly increasing from $e_0 \geqslant 0$, therefore $e_j + 1 \leqslant e_i$ and $2^{e_j-e_i} \leqslant 1/2$. We draw

$$2^{e_i}\sum_{j=0}^{i-1} 2^{e_j} \left(\frac{1}{2^{e_i}+1} + \frac{1}{\sum_{j=0}^{i-1} 2^{e_j}+1} \right) > 2\sum_{j=0}^{i-1} 2^{e_j} - \frac{i}{2}.$$

Furthermore

$$\sum_{i=1}^{r} \mathcal{A}^{\texttt{merge}}_{2^{e_i},2^{e_{i-1}}+\cdots+2^{e_0}} \geqslant \sum_{i=1}^{r} \left(2\sum_{j=0}^{i-1} 2^{e_j} - \frac{i}{2} + 1 \right)$$
$$= 2\sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} - \frac{1}{4}r(r+1) + r = 2\sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} - \frac{1}{4}r(r-3).$$

The bound is tight only when $r = 0$, that is, $n = 2^{e_0}$. We can use this inequation on equation (13.54) on the facing page:

$$\mathcal{A}(n) \geqslant \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + 2\sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} - \frac{1}{4}(\nu_n - 1)(\nu_n - 4),$$

by definition of ν. We already know how to bound the double sum in the right-hand side by mean of inequation (13.46) on page 305, therefore

$$\begin{aligned}\mathcal{A}(n) &\geqslant \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + 2(n - 2^{\lfloor \lg n \rfloor} + 2^{\nu_n - 1} - \nu_n) - \frac{1}{4}(\nu_n - 1)(\nu_n - 4)\\ &= \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + 2n - 2^{\lfloor \lg n \rfloor + 1} + 2^{\nu_n} - \frac{1}{4}(\nu_n^2 + 3\nu_n + 4).\end{aligned}$$

We can now make use of the lower bound of $\mathcal{A}(2^p)$ in inequation (13.33) on page 299, where $\alpha := \sum_{k\geqslant 0} 1/(2^k+1) \simeq 1.2644997803484442092$:

$$\begin{aligned}\mathcal{A}(n) &\geqslant \sum_{i=0}^{r} (e_i 2^{e_i} - (\alpha - 1)2^{e_i} - 1) + 2n - 2^{\lfloor \lg n \rfloor + 1} + 2^{\nu_n} - \frac{\nu_n^2 + 3\nu_n + 4}{4}\\ &= \sum_{i=0}^{r} e_i 2^{e_i} + (3 - \alpha)n - 2^{\lfloor \lg n \rfloor + 1} + 2^{\nu_n} - \frac{1}{4}(\nu_n^2 + 7\nu_n + 4).\end{aligned}$$

Note that $3 - \alpha > 0$. We already found a lower bound for the remaining sum, in inequation (13.45) on page 305, therefore

$$\mathcal{A}(n) \geqslant n\lfloor \lg n \rfloor + (3 - \alpha)n - 2^{\lfloor \lg n \rfloor + 2} + 2^{\nu_n} - (\nu_n^2 + 7\nu_n - 4)/4.$$

Let $h(x) := 2^x - (x^2 + 7x - 4)/4$ be a function over the positive real numbers. It has no positive roots and its derivative is $h'(x) = 2^x \ln 2 - (2x + 7)/4$, whose unique positive root is approximately 1.98399. We have $h(1) = 1, h(2) = 1/2$, and $h(3) = 3/2$. See FIGURE 81 on page 311, where we should mind that the minimum is not exactly $h(2)$. We deduce that, for any $n > 0$, we have $2^{\nu_n} - (\nu_n^2 + 7\nu_n - 4)/4 \geqslant 1/2$. We can weaken the lower bound accordingly:

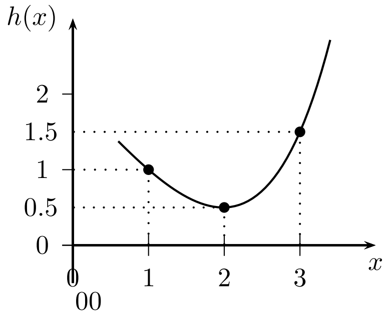
$$\mathcal{A}(n) \geqslant n\lfloor \lg n \rfloor + (3 - \alpha)n - 2^{\lfloor \lg n \rfloor + 2} + 1/2. \tag{13.55}$$

We can further weaken it to get a rough, but more legible, bound as

$$\mathcal{A}(n) > n(\lg n - 1) + (3 - \alpha)n - 4n + 1/2 = n \lg n - (\alpha + 2)n + 1/2.$$

Let us strive now to find a small upper bound of $\mathcal{A}(n)$ from equation (13.54) on the current page:

$$\mathcal{A}(n) = \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + \sum_{i=1}^{r} \frac{2^{e_i}}{2^{e_i} + 1} \sum_{j=0}^{i-1} 2^{e_j} + \sum_{i=1}^{r} \frac{2^{e_i} \sum_{j=0}^{i-1} 2^{e_j}}{\sum_{j=0}^{i-1} 2^{e_j} + 1} + r$$



1-11 001

tempcolor[rgb]blacktempcolor[rgb]white

$$\leqslant \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + \sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} + \sum_{i=1}^{r} 2^{e_i} + r$$
$$= \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + \sum_{i=1}^{r}\sum_{j=0}^{i-1} 2^{e_j} + (n - 2^{e_0}) + (\nu_n - 1).$$

We bounded the double sum in inequation (13.40) on page 303, thus

$$\mathcal{A}(n) \leqslant \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + (2n - 2^{\lfloor \lg n \rfloor + 1} - \nu_n + 1) + n + \nu_n - 2^{e_0} - 1$$
$$\leqslant \sum_{i=0}^{r} \mathcal{A}(2^{e_i}) + 3n - 2^{\lfloor \lg n \rfloor + 1} - 1.$$

The bound is tight only if $n = 1$. Using inequation (13.33) on page 299, that is, $\mathcal{A}(2^p) \leqslant p2^p$, and inequation (13.42) on page 303, that is, $\sum_{i=0}^{r} e_i 2^{e_i} \leqslant n\lfloor \lg n \rfloor$, yields

$$\mathcal{A}(n) \leqslant n\lfloor \lg n \rfloor + 3n - 2^{\lfloor \lg n \rfloor + 1} - 1. \tag{13.56}$$

Let us eliminate the integer parts and bit sums, so the asymptotic analysis becomes easier. Inequations (13.43) on page 304 yield

$$\mathcal{A}(n) \leqslant n \lg n + 3n - (n + 1) - 1 = n \lg n + 2n - 2.$$

In summary, we found the following bounds

$$n \lg n - (\alpha + 2)n + 1/2 < \mathcal{A}(n) \leqslant n \lg n + 2n - 2,$$

which are tight enough to allow us to derive the asymptotic equivalences

$$\mathcal{A}(n) \sim n \lg n \sim \mathcal{W}(n), \text{ as } n \to \infty.$$

Again, we find that this is the same as the special case $n = 2^{e_0}$.

The program, finally. It may come as a surprise that we have not written the program, except for the functions `solo/1` and `merge/2`, but we have deduced its delay nevertheless. In fact, the delays we found are not the delays of the program yet to be written, but the delays of an ideal version without `solo/1` and without dealing with all the empty lists and singletons ending many recursive calls: only `merge/2` was exactly defined and accounted for. Here are the definitions again:

```
merge(   [],      Q)            -> Q;
merge(    P,     [])            -> P;
merge([I|P],Q=[J|_]) when I < J -> [I|merge(P,Q)];
merge(    P,  [J|Q])            -> [J|merge(P,Q)].
```

```
solo(   []) -> [];
solo([I|L]) -> [[I]|solo(L)].
```

The extra delay incurred by `solo/1` is known or obvious: $\mathcal{D}_n^{\mathsf{solo}} = n + 1$. We need now to complete the program and sum the extra delays each new function entails. Let us call `level/1` the function which computes the level in the tree just above the current one, which is the argument.

```
level([P,Q|S]) -> [merge(P,Q)|level(S)];
level(      L) -> L.
```

Note how we do not need to write a specific clause handling the case when the list is empty: the singleton list and the empty list are treated likewise, which is really neat. In our previous detailed analysis, we only counted the delays of the calls to `merge/2`, therefore, we have now to find how many times in total `level/1` is called to sort n items. The definition shows that, to one call `level([P,Q|S])` corresponds one call `merge(P,Q)`, so, if a level contains k lists, then the calls total $\lfloor k/2 \rfloor + 1$. First, let us assume that we sort $n = 2^p$ items. Previous study revealed that there are 2^{p-1} merges on the leaves, then 2^{p-2} on the level above etc. until 2^0. The extra delay due to `level/1` is thus $\sum_{k=0}^{p-1} 2^k + p$, the additional p is the number of empty lists terminating each level, which trigger the second clause of `level/1`. Hence, in total, the extra delay is $2^p + p - 1$. Then, the function `level/1` must be composed with itself until one list remains in the level, corresponding to the root. It follows:

```
all([P]) -> P;
all(  L) -> all(level(L)).
```

Notice that, just as with the definition of `level/1`, the order of the clauses of `all/1` is crucial. The extra delay incurred is the number of levels in the merge tree. Since we assume that the number of items is 2^p, we already know that there are $p + 1$ levels. Finally, the merge sort function, `sort/1`, starts by creating the singletons and then merging all of them and the resulting lists as well, etc. until one list remains.

```
sort([]) -> [];
sort( L) -> all(solo(L)).
```

Here, the extra delay due to `sort/1` is simply 1. In sum, the total extra delay with respect to the ideal version of merge sort which only took into account the delay of `merge/2` is

$$(2^p + 1) + (2^p + p - 1) + (p + 1) + 1 = 2^{p+1} + 2p + 2 = 2n + 2 \lg n + 2.$$

Let us recall the delays we found earlier in the case $n = 2^p$:

$$\mathcal{B}(n) = \tfrac{1}{2} n \lg n + n - 1;\ \mathcal{W}(n) = n \lg n;$$

$$n \lg n - (\alpha - 1)n - 1 < \mathcal{A}(n) \leqslant n \lg n.$$

Let us add them the extra delay we just deduced:

$$\mathcal{B}_n^{\mathsf{sort}} = \mathcal{B}(n) + (2n + 2\lg n + 2) = \tfrac{1}{2} n \lg n + 3n + 2\lg n + 1;$$
$$\mathcal{W}_n^{\mathsf{sort}} = \mathcal{W}(n) + (2n + 2\lg n + 2) = n \lg n + 2n + 2\lg n + 2;$$
$$n \lg n + (3 - \alpha)n + 2\lg n + 1 < \mathcal{A}_n^{\mathsf{sort}} \leqslant n \lg n + 2n + 2\lg n + 2.$$

It is important to check these involved calculations against some experiments. Let us thread an integer counter into all function calls and run a few tests. We do not need to transform the previous program into tail form. Consider the following method, consisting in handling each inner call separately and projecting their result in order to separate the value and the counter, then reconstructing the next call or the final value.

```
-module(ms_c).
-compile(export_all).       % So we can test all functions

sort([]) -> {[],1};
sort( L) -> {S,C}=solo(L,1), all(S,C).

all([P],C) -> {P,C+1};
all(  L,C) -> {S,D}=level(L,C+1), all(S,D).

level([P,Q|S],C) -> {R,D}=merge(P,Q,C+1), {T,E}=level(S,D),
                    {[R|T],E};
level(      L,C) -> {L,C+1}.

merge(   [],      Q,C)            -> {Q,C+1};
merge(    P,     [],C)            -> {P,C+1};
merge([I|P],Q=[J|_],C) when I < J -> {M,D}=merge(P,Q,C+1),
                                     {[I|M],D};
merge(    P,  [J|Q],C)            -> {M,D}=merge(P,Q,C+1),
                                     {[J|M],D}.

solo(   [],C) -> {[],C+1};
solo([I|L],C) -> {S,D}=solo(L,C+1), {[[I]|S],D}.
```

Erlang permits simplifying the syntax of the `case` constructs with only one clause: instead of

`case` *Exp*$_1$ `of` *Pattern* `->` *Exp*$_2$ `end`

we can equivalently write

Pattern `=` *Exp*$_1$, *Exp*$_2$.

Now we can check the best case of FIGURE 73 on page 296:

$\texttt{ms\_c:sort([1,2,3,4,5,6,7,8])} \twoheadrightarrow \texttt{\{[1,2,3,4,5,6,7,8],43\}}$,

which means $\texttt{ms:sort([1,2,3,4,5,6,7,8])} \xrightarrow{43} \texttt{[1,2,3,4,5,6,7,8]}$, where `ms` is the original module without the counter, so this is in accordance with the prediction $\mathcal{B}_8^{\mathsf{sort}} = 8 \cdot 3/2 + 3 \cdot 8 + 2 \cdot 3 + 1 = 43$. Similarly, the worst case in FIGURE 75 on page 298 leads to the test

$\texttt{ms\_c:sort([7,3,5,1,4,8,6,2])} \twoheadrightarrow \texttt{\{[1,2,3,4,5,6,7,8],48\}}$,

which means $\texttt{ms:sort([7,3,5,1,4,8,6,2])} \xrightarrow{48} \texttt{[1,2,3,4,5,6,7,8]}$, in accordance with the instance $\mathcal{W}_8^{\mathsf{sort}} = 8 \cdot 3 + 2 \cdot 8 + 2 \cdot 3 + 2 = 48$. We could further use the program we designed in chapter on page 195 to generate all permutations of `[1,2,3,4,5,6,7,8]`, feed them to merge sort and take the arithmetic mean of the delays. We wrote

```
-module(perm).
-compile(export_all).

perm(   []) -> [];
perm(  [I]) -> [[I]];
perm([I|L]) -> dist(I,perm(L)).

dist(_,          []) -> [];
dist(I,[Perm|Perms]) -> join(ins(I,Perm),dist(I,Perms)).

ins(I,        []) -> [[I]];
ins(I,Perm=[J|L]) -> [[I|Perm]|push(J,ins(I,L))].

push(_,          []) -> [];
push(I,[Perm|Perms]) -> [[I|Perm]|push(I,Perms)].

join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

Now we can write a module `ms_ave` which exports a function `means/1` such that `means`(n) rewrites in the list of the mean values of `ms:sort/1` from n down to 1.

```
-module(ms_ave).
-export([means/1]).

means(0) -> [];
means(N) -> [mean(N)|means(N-1)].
```

```
mean(N) -> ave(sort(perm:perm(init(N)))).

sort(          []) -> [];
sort([Perm|Perms]) -> [snd(ms_c:sort(Perm))|sort(Perms)].

init(0) -> [];
init(N) -> [N|init(N-1)].

ave(L) -> {Sum,Len}=sum_len(L,0,0), Sum/Len.

sum_len(   [],Sum,Len) -> {Sum,Len};
sum_len([N|L],Sum,Len) -> sum_len(L,Sum+N,Len+1).

snd({_,Y}) -> Y.
```

Module ms_ave depends upon independent modules ms_c and perm, so these must be compiled first. Notive how we interleaved the computations of the sum and the length in one function, ms_ave:sum_len/1, performing only one traversal of the list of permutations and how, being in tail form, it uses a small, constant amount of the control stack. A possible session under the Erlang shell:

```
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2]
 [async-threads:0] [kernel-poll:false]

Eshell V5.6.3  (abort with ^G)
1> c(ms_c).
{ok,ms_c}
2> c(perm).
{ok,perm}
3> c(ms_ave).
{ok,ms_ave}
4> ms_ave:means(10).
[62.84444444444444,56.62222222222222,46.733333333333334,
 40.733333333333334,34.93333333333333,29.466666666666665,
 21.666666666666668,16.666666666666668,10.0,4.0]
5> ▯
```

Since we expect the resulting numbers to be rational, with the help of some *computer algebra system*, we can find the exact expression of these numbers in FIGURE 82 on page 316. Erlang has the peculiarity that integer arithmetic is implicitly performed with arbitrary precision, so we can compute integers as big as the computer memory can hold. Usually, programming languages require a dedicated library to do this

n	1	2	3	4	5	6	7	8	9	10
`mean(`n`)`	4	10	$\frac{50}{3}$	$\frac{65}{3}$	$\frac{442}{15}$	$\frac{524}{15}$	$\frac{611}{15}$	$\frac{701}{15}$	$\frac{2548}{45}$	$\frac{2828}{45}$

FIGURE 82: Values of `mean(`n`)` for $1 \leqslant n \leqslant 10$

and the integers are mapped by the run-time system to the hardware integers, whose representation may lead to underflows and overflows. In this respect, Erlang is quite a high-level language. We have to compare now `mean(`n`)` with $\mathcal{A}_n^{\mathsf{sort}}$. We can compute exactly the latter with help of equation (13.32) on page 298:

$$\mathcal{A}(2^p) = p2^p + 2^p - 1 - 2^p \sum_{k=0}^{p-1} \frac{1}{2^k+1}.$$

Therefore,

$$\mathcal{A}_{2^p}^{\mathsf{sort}} = \mathcal{A}(2^p) + (2^{p+1} + 2p + 2) = 2^p \left(p + 3 - \sum_{k=0}^{p-1} \frac{1}{2^k+1} \right) + 2p + 1.$$

This yields the comfort to know that

$$\mathcal{A}_2^{\mathsf{sort}} = 2\left(1 + 3 - \frac{1}{1+1}\right) + 2 \cdot 1 + 1 = 10 = \texttt{mean(2)};$$
$$\mathcal{A}_4^{\mathsf{sort}} = 4\left(2 + 3 - \frac{1}{2} - \frac{1}{2+1}\right) + 2 \cdot 2 + 1 = \frac{65}{3} = \texttt{mean(4)};$$
$$\mathcal{A}_8^{\mathsf{sort}} = 8\left(3 + 3 - \frac{1}{2} - \frac{1}{3} - \frac{1}{4+1}\right) + 2 \cdot 3 + 1 = \frac{701}{15} = \texttt{mean(8)}.$$

Now, what if n is not a power of 2? Let us assume that $n \neq 2^p$ is expressed as a binary number with m bits as $(b_{m-1}b_{m-2} \ldots b_1b_0)_2$, that is, $n := \sum_{i=0}^{m-1} b_i 2^i$. We have shown by equation (13.35) on page 301 that $m = \lfloor \lg n \rfloor + 1$. Let L_n be the contribution of the arrows of function `level/1`. As we pointed out earlier, if level k contains $I(k)$ items, there are $\lfloor I(k)/2 \rfloor + 1$ calls to `level/1`, so remains to determine $I(k)$. A look back at the example in FIGURE 77 on page 300, where $n = 7$, helps us to guess the answer:

$$I(k+1) = \lceil I(k)/2 \rceil \text{ and } I(0) = n,$$

meaning that $k = 0$ represents the level of the leaves. This recurrent definition is equivalent to the closed form $I(k) = \lceil n/2^k \rceil$ because $\lceil \lceil x \rceil / q \rceil = \lceil x/q \rceil$. The proof is as follows. The equality is equivalent to the conjunction of the two complementary inequalities $\lceil \lceil x \rceil / q \rceil \geqslant \lceil x/q \rceil$ and $\lceil \lceil x \rceil / q \rceil \leqslant \lceil x/q \rceil$. The former is straightforward because it is a consequence of $\lceil y \rceil \geqslant y$, for any real number y: $\lceil x \rceil \geqslant x \Rightarrow \lceil x \rceil / q \geqslant x/q \Rightarrow$

$\lceil\lceil x\rceil/q\rceil \geqslant \lceil x/q\rceil$. Next, because both sides of the inequality are integers, $\lceil\lceil x\rceil/q\rceil \leqslant \lceil x/q\rceil$ is equivalent to state that $p \leqslant \lceil\lceil x\rceil/q\rceil \Rightarrow p \leqslant \lceil x/q\rceil$, for any integer p. An obvious lemma is that if i is an integer and y a real number, $i \leqslant \lceil y\rceil \Leftrightarrow i \leqslant y$, so the original inequality is transitively equivalent to $p \leqslant \lceil x\rceil/q \Rightarrow p \leqslant x/q$, for any integer p, which is trivially equivalent to $pq \leqslant \lceil x\rceil \Rightarrow pq \leqslant x$. This implication is true from the same lemma above, achieving the proof. In the end, $L_n = \sum_{k=0}^{?} \left(\lfloor\lceil n/2^k\rceil/2\rfloor + 1\right)$. What is the greatest value of k? This is where the hypothesis $n \neq 2^p$ comes into play. If we compare figure 77 on page 300 with figure 72 on page 294, where $n = 2^3$, we find that $k = m$ for the former and $k = m-1$ for the latter. Therefore, if $n \neq 2^p$,

$$L_n = \sum_{k=0}^{m} \left\lfloor \frac{1}{2}\left\lceil \frac{n}{2^k}\right\rceil \right\rfloor + (m+1) = \sum_{k=0}^{m} \left\lfloor \frac{1}{2}\left\lceil \frac{n}{2^k}\right\rceil \right\rfloor + \lfloor \lg n\rfloor + 2.$$

For all integers q, $q = \lfloor q/2\rfloor + \lceil q/2\rceil$ holds, hence $q = \lceil n/2^k\rceil$ yields

$$\begin{aligned} L_n &= \sum_{k=0}^{m} \left(\left\lceil \frac{n}{2^k}\right\rceil - \left\lceil \frac{1}{2}\left\lceil \frac{n}{2^k}\right\rceil\right\rceil \right) + \lfloor \lg n\rfloor + 2 \\ &= \sum_{k=0}^{m}\left\lceil \frac{n}{2^k}\right\rceil - \sum_{k=0}^{m}\left\lceil \frac{n}{2^{k+1}}\right\rceil + \lfloor \lg n\rfloor + 2 = \left\lceil \frac{n}{2^0}\right\rceil - \left\lceil \frac{n}{2^{m+1}}\right\rceil + \lfloor \lg n\rfloor + 2 \\ &= n + \lfloor \lg n\rfloor + 2. \end{aligned}$$

As far as `all/1` is concerned, we determined previously that it contributes as much as the number of levels in the merge tree, which is here $m + 1 = \lfloor \lg n\rfloor + 2$. We have the extra 1 due to `sort/1` and the $n + 1$ from `solo/1`. In sum, the additional delay to reach the actual implementation delay is

$$(n + \lfloor \lg n\rfloor + 2) + (\lfloor \lg n\rfloor + 2) + 1 + (n + 1) = 2n + 2\lfloor \lg n\rfloor + 6.$$

Let us remind of the intervals for the delays of the ideal version of merge sort and add this extra time to get the actual intervals:

$$\begin{aligned} \mathcal{B}_n^{\mathsf{sort}} &\geqslant n\lfloor \lg n\rfloor/2 + 4n - 2^{\lfloor \lg n\rfloor+1} + 2\lfloor \lg n\rfloor + 5, \\ \mathcal{B}_n^{\mathsf{sort}} &\leqslant n\lfloor \lg n\rfloor/2 + 5n - 2^{\lfloor \lg n\rfloor+1} + 2\lfloor \lg n\rfloor + 5; \\ \mathcal{W}_n^{\mathsf{sort}} &\geqslant n\lfloor \lg n\rfloor + 4n - 2^{\lfloor \lg n\rfloor+2} + 2\lfloor \lg n\rfloor + 8, \\ \mathcal{W}_n^{\mathsf{sort}} &\leqslant n\lfloor \lg n\rfloor + 5n - 2^{\lfloor \lg n\rfloor+1} + 2\lfloor \lg n\rfloor + 5; \\ \mathcal{A}_n^{\mathsf{sort}} &\geqslant n\lfloor \lg n\rfloor + (5-\alpha)n - 2^{\lfloor \lg n\rfloor+2} + 2\lfloor \lg n\rfloor + 13/2, \\ \mathcal{A}_n^{\mathsf{sort}} &\leqslant n\lfloor \lg n\rfloor + 5n - 2^{\lfloor \lg n\rfloor+1} + 2\lfloor \lg n\rfloor + 5. \end{aligned} \tag{13.57}$$

The derivations ensure that these delays are valid for any integer $n > 0$.

$n =$	3	5	6	7	9	10
`mean`$(n) \simeq$	16.66	29.46	34.93	40.73	56.62	62.84
lower $\mathcal{A}_n^{\mathsf{sort}} \simeq$	14.70	23.17	28.91	34.64	41.11	47.85
upper $\mathcal{A}_n^{\mathsf{sort}} =$	21.00	36.00	43.00	50.00	67.00	75.00

FIGURE 83: Checking average delay of merge sort when $n \neq 2^p$

It is complicated to caracterise the best and worst cases but we can put to the test the average delay because we already have some exact values from instrumenting the actual code by adding a counter and a permutation generator. Let us use Erlang to compute the bounds:

```
-module(mean).
-export([bounds/1]).

floor(X) when X < trunc(X) -> trunc(X) - 1;
floor(X)                   -> trunc(X).

log2(X) -> math:log(X)/math:log(2).

exp2(0) -> 1;
exp2(N) -> E=exp2(N div 2), (1 + N rem 2)*(E*E).

bounds(N) -> A=1.26449978034844420919131974725549848255770,
             LOG=floor(log2(N)), EXP=exp2(LOG),
             {N*LOG + (5-A)*N - 4*EXP + 2*LOG + 13/2,
              N*LOG +     5*N - 2*EXP + 2*LOG +    5}.
```

Function `trunc/1` is predefined. Module `math` is also predefined and `math:log/1` implements the Napierian logarithm. The approximate value of α can be compute in Erlang, but we preferred to use a computer algebra system. Note how we efficiently computed the binary exponentiation 2^n by means of the recurrent equations

$$2^0 = 1, \quad 2^{2m} = (2^m)^2, \quad 2^{2m+1} = 2(2^m)^2. \tag{13.58}$$

The delay $\mathcal{D}_n^{\mathsf{exp2}}$ thus satisfies the equations

$$\mathcal{D}_0^{\mathsf{exp2}} = 1; \quad \mathcal{D}_n^{\mathsf{exp2}} = 1 + \mathcal{D}_{\lfloor n/2 \rfloor}^{\mathsf{exp2}}, \text{ if } n > 0.$$

Therefore, if $n > 0$, it is 1 plus the number of bits of n, that is, $\mathcal{D}_n^{\mathsf{exp2}} = \lfloor \lg n \rfloor + 2$, else $\mathcal{D}_0^{\mathsf{exp2}} = 1$. We can now complete the table of the bounds of $\mathcal{A}_n^{\mathsf{sort}}$ and check them against the experimental results `mean`(n) in FIGURE 83 on page 319.

Stability. This implementation of merge sort is unstable because `merge/2` does not preserves the relative order of equal numbers. In order to obtain stability, we should write instead

```
merge(   [],      Q)            -> Q;
merge(    P,     [])            -> P;
merge(P=[I|_],[J|Q]) when I > J -> [J|merge(P,Q)];        % Here
merge(  [I|P],    Q)            -> [I|merge(P,Q)].
```

This works because the first list we pass to `merge/2` always contains items which were located *before* the items of the second list.

Improvement. It is easy to improve our implementation of merge sort by avoiding the construction of the initial list of singletons, that is, the leaves of the merge tree, and instead build directly the lists with two items, *without using* `merge/2`. Let us name the function in charge of this `duo/1`. The program is as follows:

```
-module(ms_opt).
-export([sort/1]).

sort([]) -> [];
sort( L) -> all(duo(L)).

duo([I,J|L]) when I > J -> [[J,I]|duo(L)];
duo([I,J|L])            -> [[I,J]|duo(L)];
duo(      L)            -> [L].

all([P]) -> P;
all(  L) -> all(level(L)).

level([P,Q|S]) -> [merge(P,Q)|level(S)];
level(      L) -> L.

merge(   [],      Q)            -> Q;
merge(    P,     [])            -> P;
merge(P=[I|_],[J|Q]) when I > J -> [J|merge(P,Q)];
merge(  [I|P],    Q)            -> [I|merge(P,Q)].
```

This simple modification saves $\mathcal{D}_n^{\texttt{solo}} = n+1$ function calls and the first call to `all/1` and `level/1`, that is, 1 for the call to `all/1`, plus $\lfloor n/2 \rfloor$ calls to `merge/2` on pairs of singletons, plus 1 for the last singleton or the empty list in `level/1`, that is, $\lfloor n/2 \rfloor \mathcal{D}_{1,1}^{\texttt{merge}} + 2 = 2\lfloor n/2 \rfloor + 2$. Instead, we have the delay of `duo/2`, that is, an additional delay of $\lfloor n/2 \rfloor + 1$.

The total delay is hence decreased by

$$(n+1+2\lfloor n/2\rfloor+2)-(\lfloor n/2\rfloor+1)=n+\lfloor n/2\rfloor+2.$$

By modifying module `ms_opt` so that all functions thread a counter and by testing some cases we already know, we can reinforce our conviction that our deduction is correct.

Higher-order merge sort. Just as we made insertion sort able to sort any kind of data as long as it is totally ordered by a supplied function, we can make merge sort by threading the comparison:

```
-module(ms_ho).
-export([sort/2]).

sort( _,[]) -> [];
sort(Gt, L) -> all(Gt,duo(Gt,L)).

duo(Gt,[I,J|L]) -> case Gt(I,J) of
                     true  -> [[J,I]|duo(Gt,L)];
                     false -> [[I,J]|duo(Gt,L)]
                   end;
duo( _,      L) -> [L].

all( _,[P]) -> P;
all(Gt,  L) -> all(Gt,level(Gt,L)).

level(Gt,[P,Q|S]) -> [merge(Gt,P,Q)|level(Gt,S)];
level( _,      L) -> L.

merge( _,     [],      Q) -> Q;
merge( _,      P,     []) -> P;
merge(Gt,P=[I|R],Q=[J|S]) -> case Gt(I,J) of
                               true  -> [J|merge(Gt,P,S)];
                               false -> [I|merge(Gt,R,Q)]
                             end.
```

Now the following calls become possible:

`ms_ho:sort(fun(I,J)->I>J end,[4,3,1,0,2])` $\twoheadrightarrow$ `[0,1,2,3,4]`;
`ms_ho:sort(fun(I,J)->I<J end,[4,3,1,0,2])` $\twoheadrightarrow$ `[4,3,2,1,0]`.

Stability is useless if the only type of data to be sorted is integer, so let us take an association list whose keys are integers but data is not:

```
ms_ho:sort(fun({I,_},{J,_}) -> I > J end,
           [{6,six},{1,one},{6,seis},{2,two}]).
```

The result is `[{1,one},{2,two},{6,six},{6,seis}]` because the relative order of `{6,six}` and `{6,seis}` is preserved.

Comparison with insertion sort. We introduce Landau's notation, which we define as

$$f(n) \in o(g(n)) \Leftrightarrow \lim_{n\rightarrow\infty} f(n)/g(n) = 0. \tag{13.59}$$

Equivalence (10.17) on page 238 imply $\lim_{n\rightarrow\infty}(\lg n)/n = 0$, which we can equivalently express with Landau's notation as $\lg n \in o(n)$. Thus, $n \lg n \in o(n^2)$. Moreover, a consequence of inequations (10.16) on page 238 is that the average delay $\mathcal{A}_n^{\mathtt{i2wb}}$ of the two-way insertion sort is lower-bounded as follows:

$$\tfrac{1}{8}(n^2 - 13n + \ln n - 10 + \ln 2) < \mathcal{A}_n^{\mathtt{i2wb}}.$$

In this chapter, we established inequation (13.57) on the next page about the upper bound of merge sort, which we can weaken a little bit to make it more wieldy:

$$\begin{aligned}\mathcal{A}_n^{\mathtt{sort}} &\leqslant n\lfloor\lg n\rfloor + 5n - 2^{\lfloor\lg n\rfloor+1} + 2\lfloor\lg n\rfloor + 5\\ &\leqslant n\lg n + 4n + 2\lg n + 4.\end{aligned}$$

Therefore, we expect the existence of $n_0 > 0$, such that, for all $n \geqslant n_0$,

$$\mathcal{A}_n^{\mathtt{sort}} < \mathcal{A}_n^{\mathtt{i2wb}}.$$

The problem is to determine n_0, that is, we want to solve the inequation

$$n\lg n + 4n + 2\lg n + 4 < \tfrac{1}{8}(n^2 - 13n + \ln n - 10 + \ln 2).$$

This is where a computer algebra system comes handy to deduce that it is equivalent to $n \geqslant 100$, that is, $n_0 = 100$. Since we worked on bounds of the delays, this result does not mean that the crossing of the averages happens at $n = 100$, only that we are certain that it happened whenever $n \geqslant 100$. In order to get a lower bound, let us consider the dual situation: we want the upper bound of $\mathcal{A}_n^{\mathtt{i2wb}}$ to be lower than the lower bound of $\mathcal{A}_n^{\mathtt{sort}}$. To make the point clearer, we still consider the fastest version of insertion sort and the slowest version of merge sort we found. Another implication of (10.16) on page 238 is

$$\mathcal{A}_n^{\mathtt{i2wb}} < \tfrac{1}{8}(n^2 - 13n + \ln n - 10 + \ln 2 + 7).$$

Furthermore, we found on the preceding page a lower bound we can now weaken as usual:

$$\begin{aligned}\mathcal{A}_n^{\mathtt{sort}} &> n\lfloor\lg n\rfloor + (5-\alpha)n - 2^{\lfloor\lg n\rfloor+2} + 2\lfloor\lg n\rfloor + 13/2\\ &> n\lg n - \alpha n + 2\lg n + 5/2.\end{aligned}$$

With a computer algebra system, we obtain

$$1 \leqslant n \leqslant 50 \Rightarrow \mathcal{A}_n^{\mathtt{i2wb}} < \mathcal{A}_n^{\mathtt{sort}}.$$

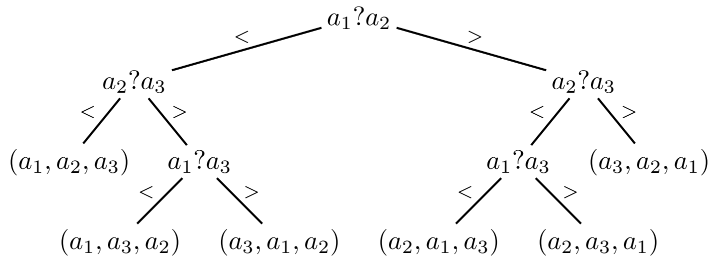


FIGURE 84: A comparison tree for sorting three items

In other words, insertion sort is certainly faster in average than merge sort for less than 50 items and it is certainly slower in average for more than 100 items. We do not have enough precision to decide in-between. Anyhow, this shows that, when comparing the asymptotic delays of two sorting algorithms, we should not conclude of the superiority of one over the other for small inputs. Also, this result entices us to drop the function `solo/2` (even `duo/2`) in favour of a function which constructs chunks of 50 items from the original list, then sort them using balanced two-way insertions and, finally, if there are more than 50 items, start merging these sorted lists. In reference to the example in FIGURE 77 on page 300, this improvement amounts to not constructing the first 50 levels in the tree, starting from the leaves, and instead use two-way insertion sort to directly build the 50th level.

Minimum-comparison sorting. In several instances, we claimed that a good sorting algorithm has a worst delay asymptotically equivalent to $n \lg n$, without any justification. In order to substantiate this statement, we need to investigate *minimum-comparison sorting*. First, it is important to realise that there are sorting algorithms which do not rely on comparisons, although they are not suited for being implemented in purely functional languages. Second, for the sake of simplicity, we shall restrict the present theoretical discussion to *distinct items*. We model the outcome of sorting n distinct objects a_1, a_2, ..., a_n, as a permutation of $(a_1, \ldots, a_n)$. There are $n!$ possible sorts, as we found out in chapter 9 on page 195. For instance, let us represent in the tree of FIGURE 84 on the facing page all the comparisons for sorting three numbers. The *external nodes* are all the permutations of (a_1, a_2, a_3). The *internal nodes* are comparisons between two items, say, in general, a_i is compared to a_j and noted $a_i?a_j$. If $a_i < a_j$, then this property holds everywhere in the left subtree, otherwise $a_i > a_j$ holds in the right subtree. Note that this tree is one possible amongst many: it corresponds to an algorithm which starts by comparing a_1 and a_2 and there are, of course, many other strategies. It does not perform redundant comparisons, though. Figure 85 on page 324 shows an excerpt of a comparison tree with such a useless comparison. The special external node $\bot$ corresponds to no permutation because the comparison $a_1 < a_3$ cannot fail as it is implied by transitivity of the previous comparisons on the path from the root. *A comparison tree for n items without redundancy has $n!$ external nodes.* Because we are investigating minimum-comparison sorting, we shall consider henceforth comparison trees with $n!$ external nodes. Furthermore, amongst them, we want to determine the trees such that the maximum number of comparisons is minimum.

The maximality constraint means that we must consider the longest paths from the root to a leaf (a leaf is an internal node whose two children are external nodes, see definitions on page 18) because the number of comparison nodes along those paths is an upper bound for sorting all the permutations. For example, in figure 84, there are four paths satisfying this maximality constraint and they contain 3 comparisons each, for example, $(a_1 < a_2) \rightarrow (a_2 < a_3) \rightarrow (a_1 < a_3)$. Let us note $S(n)$ the number of nodes on a maximum path, which is usually called the *height* of the tree, for example, $S(3) = 3$ in figure 84. The minimality constraint in the problem statement above then means that *we want a lower bound on the height of a comparison tree with $n!$ external nodes.*

A *complete binary tree* is a binary tree whose internal nodes have children which are either two internal nodes or two external nodes. If such a tree has height h, then it has 2^h external nodes. For instance, figure 86 on page 325 shows the case where the height h is 3 and there are indeed $2^h = 8$ external nodes, figured as $\Box$. Since we know that the minimum-comparison trees have $n!$ external nodes and height $S(n)$, they must contain less external nodes than a complete binary

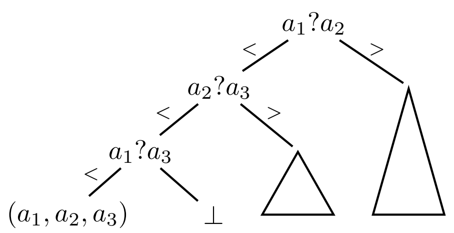


figure 85: Comparison of a_1 to a_3 is useless

tree of identical height, that is, the following holds:

$$n! \leqslant 2^{S(n)}.$$

This inequality implies the following (perhaps best read leftward):

$$\sum_{k=1}^{n} (\lfloor \lg k \rfloor + 1) - n = \sum_{k=1}^{n} \lfloor \lg k \rfloor \leqslant \sum_{k=1}^{n} \lg k = \lg n! \leqslant \lg\left(2^{S(n)}\right) = S(n).$$

The reason for manifesting the term $\lfloor \lg k \rfloor + 1$ is that its value is the number of bits of k, so the remaining sum is the total number of bits of the integers from 1 to n. The table in FIGURE 87 on page 326 shows the enumeration of the first integers in binary. The greatest power of 2 smaller than n is $2^{\lfloor \lg n \rfloor}$ because it is the number $(10\ldots0)_2$ with the same number of bits as n (so it appears in the same section of the table as n). The trick consists in counting the bits in columns, from top to bottom, and leftward. In the rightmost column, we find n bits. In the second column, from the right, we find $n - 2^1 + 1$ bits. The third from the right contains $n - 2^2 + 1$ bits etc. until the leftmost column containing $n - 2^{\lfloor \lg n \rfloor} + 1$ bits. The total number of bits in the table is

$$\begin{aligned}\sum_{k=1}^{n}(\lfloor \lg k \rfloor + 1) = \sum_{k=0}^{\lfloor \lg n \rfloor} (n - 2^k + 1) &= (n+1)(\lfloor \lg n \rfloor + 1) - 2^{\lfloor \lg n \rfloor + 1} + 1 \\ &\geqslant (n+1)(\lfloor \lg n \rfloor + 1) - 2n + 1 = n\lfloor \lg n \rfloor - n + \lfloor \lg n \rfloor + 3.\end{aligned}$$

We can finally weaken a little bit the lower bound of $S(n)$ as follows:

$$n \lg n - 3n + \lg n + 2 < n\lfloor \lg n \rfloor - 2n + \lfloor \lg n \rfloor + 3 \leqslant S(n),$$

because $x - 1 < \lfloor x \rfloor$. This proves that any optimal sorting algorithm, solely based on comparisons, requires at least $n\lfloor \lg n \rfloor - 2n$ comparisons to sort any list of n items. Since we found in inequation (13.53) on page 308 that the worst delay of merge sort, $\mathcal{W}(n)$, satisfies

$$\mathcal{W}(n) \leqslant n \lg n + 2n - 2$$

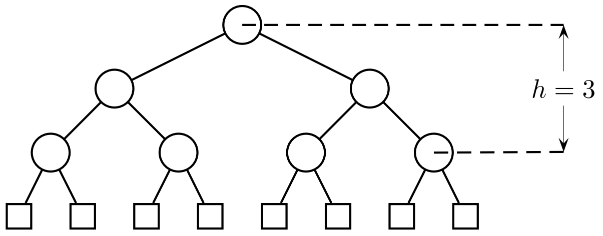


FIGURE 86: Complete binary trees of height h have 2^h external nodes.

2^0	1
2^1	10
	11
2^2	100
	101
	110
	111
2^3	1000
	1001
	1010
	1011
	1100
	1101
	1110
	1111
$\vdots$	$\vdots$
$2^{\lfloor \lg n \rfloor}$	$\ldots$
	$\vdots$
	n

FIGURE 87: Binary numbers from 1 to n

and, since $S(n) \leqslant \mathcal{W}(n)$, we now have converging bounds for the so-called *minimax* delay:

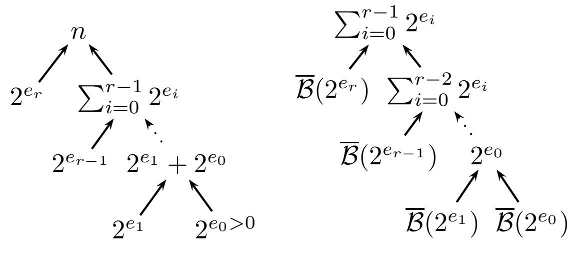
$$n \lg n - 3n + \lg n + 2 < S(n) \leqslant n \lg n + 2n - 2$$

and thus $S(n) \sim n \lg n \sim \mathcal{W}(n) \sim \mathcal{A}(n)$. This means that merge sort is, in the worst case and asymptotically, as good as any optimal sorting algorithm. However, this does not mean that it cannot be beaten by another algorithm for small values of n or in average.

Exercises. [See answers page 385.]

1. A theoretical way to compare sorting algorithms based on comparisons is to compare only the number of comparisons they perform. Determine $\overline{\mathcal{B}}(n)$, $\overline{\mathcal{W}}(n)$ and $\overline{\mathcal{A}}(n)$, which are, respectively, the best, worst and average number of comparisons in the ideal merge sort algorithm. Deduce a better upper bound for $S(n)$.
2. Find better approximations of $\mathcal{B}(n)$ and $\mathcal{W}(n)$ when $n = 2^p - 1$. Compare $\overline{\mathcal{B}}(2^p - 1)$ with $\overline{\mathcal{B}}(2^p)$ and $\overline{\mathcal{W}}(2^p - 1)$ with $\overline{\mathcal{W}}(2^p)$.

3. When looking for a lower bound of $S(n)$, we used the inequality $\sum_{k=1}^{n} \lfloor \lg k \rfloor < \sum_{k=1}^{n} \lg k$, which is quite rough. If we allow ourselves to work temporarily on real numbers, we can harvest the fruits of real analysis as follows. Let two integers $a \leqslant b$ and $f : [a, b] \rightarrow \mathbb{R}$ a real-valued, increasing and Riemann-integrable function. Then
$$\sum_{k=a}^{b-1} f(k) \leqslant \int_a^b f(x)\,dx \leqslant \sum_{k=a+1}^{b} f(k).$$
Assume this theorem and derive a better lower bound for $S(n)$.
4. The derivation of simple bounds is not as accurate as we would like because we made over-pessimistic assumptions. For instance, the inequalities
$$\mathcal{W}(n) \geqslant n\lfloor \lg n \rfloor + 2n - 2^{\lfloor \lg n \rfloor + 2} + 2 > n(\lg n - 1) + 2n - 4n + 2$$
rely on $\lfloor \lg n \rfloor > \lg n - 1$ and $n \geqslant 2^{\lfloor \lg n \rfloor}$. The latter is tight if n is a power of 2. But if it is so, we could have used an equality in stead of the former inequality: $\lfloor \lg n \rfloor = \lg n$. Prove the finer
$$\begin{aligned} \tfrac{1}{2} n \lg n \leqslant \mathcal{B}(n) &\leqslant \tfrac{1}{2} n \lg n + \tfrac{3}{2} n - 1, \\ n \lg n - 2n + 2 \leqslant \mathcal{W}(n) &\leqslant n \lg n + \tfrac{3}{2} n - 1, \\ n \lg n - (\alpha + 1)n + \tfrac{1}{2} \leqslant \mathcal{A}(n) &\leqslant n \lg n + \tfrac{3}{2} n - 1. \end{aligned}$$
Using Question 1, find a smaller upper bound for $S(n)$.
Hint. In general, we want a lower bound such that $n \lg n + \lambda n \leqslant n\lfloor \lg n \rfloor + \psi n - \omega 2^{\lfloor \lg n \rfloor}$, with $\psi > 0$, $\omega > 0$ and λ is maximum. In the example above, $\psi = 2$ and $\omega = 4$ (the constant term can be cancelled on both sides) and $\lambda = -2$. The dual case for upper bounds is obtained by reversing the comparison and minimising λ. Let us allow n to range over the real numbers and represent it in the mantissa-exponent notation in base 2, that is, we set $n = q2^p$, where $p \in \mathbb{N}, q \in \mathbb{R}$ and $1 \leqslant q < 2$. Therefore $\lg n = p + \lg q$, with $0 \leqslant \lg q < 1$, and thus $\lfloor \lg n \rfloor = p$. Use this representation in the inequality, study the resulting real-valued function of q and maximise λ on the real interval $[1, 2]$, depending on conditions upon ψ and ω.
5. Implement in two Erlang clauses the function ν (bit sum).
6. Let ρ_n the largest power of 2 dividing n. In other words, $\rho_n = 2^{e_0}$ if we let $n = 2^{e_r} + \cdots + 2^{e_1} + 2^{e_0} > 0$, with $e_r > \cdots > e_1 > e_0 \geqslant 0$ and $r \geqslant 0$. This function is called the *ruler function* and can also be understood as computing the number of trailing zeros in the binary representation of n. Define recursively ρ_n and prove
$$\rho_n = 1 + \nu_{n-1} - \nu_n,$$


n
2^{e_r}
$\sum_{i=0}^{r-1} 2^{e_i}$
$2^{e_{r-1}}$
$2^{e_1} + 2^{e_0}$
2^{e_1}
$2^{e_0 > 0}$
$\sum_{i=0}^{r-1} 2^{e_i}$
$\overline{\mathcal{B}}(2^{e_r})$
$\sum_{i=0}^{r-2} 2^{e_i}$
$\overline{\mathcal{B}}(2^{e_{r-1}})$
2^{e_0}
$\overline{\mathcal{B}}(2^{e_1})$
$\overline{\mathcal{B}}(2^{e_0})$

(a) Nodes with list lengths (b) Nodes with best delays

FIGURE 88: Best delay mergings for n even

where ν_n is the sum of the bits of n. Finally conclude that

$$\sum_{k=1}^{n} \rho_k = n - \nu_n \sim n, \text{ as } n \to \infty.$$

7. Let $\overline{\mathcal{B}}(n)$ be the number of comparisons in the best case of the ideal merge sort. Prove what Question 2 suggested:

$$\overline{\mathcal{B}}(n+1) = \overline{\mathcal{B}}(n) + \nu_n.$$

Examine vertically FIGURE 87 on page 326 and deduce simple bounds on $\sum_{k=0}^{n-1} \nu_k$. (Note that the row $k = 0$ is missing in the table but it matters even though $\nu_0 = 0$.) Deduce bounds on $\overline{\mathcal{B}}(n)$ and compare them with the ones in Question 1 and 4.

Hint. Let us consider n expressed as a sum of powers of 2: $n := \sum_{i=0}^{r} 2^{e_i}$. When n is even, $e_0 > 0$ and all the merge trees in their best cases are isomorphic to the tree displayed in FIGURE 88a on page 328, whose nodes contain the lengths of the lists in the corresponding merge tree. They are also isomorphic to the tree in FIGURE 88b on page 328, whose nodes contain the best delays of the corresponding merges in the merge tree. The total best delay $\overline{\mathcal{B}}(n)$ is the sum of the delays of all the nodes. The complementary situation, n odd, means that n can be written in binary as $\sigma 0 1^q$, where σ is some bit sequence and q some positive integer. In terms of the decomposition into powers of 2, this is equivalent to say that there exists a positive q such that $e_q > q$ and, for all $i < q$, $e_i = i$. This leads to the trees in FIGURE 89 on page 329. Make the trees for $n+1$ from the cases "n even" and "n odd." Conclude that $\overline{\mathcal{B}}(n+1) = \overline{\mathcal{B}}(n) + \nu_n$, whatever the parity of n is. Does the same recurrent equation hold for $\overline{\mathcal{W}}$, as Question 2 suggested?

8. We mentionned that the beauty of our implementation of merge sort is the fact that no provision is made for handling the un-

$$\begin{tikzcd}[column sep=tiny]
 & \sum_{j=q}^{r} 2^{e_j} + \sum_{j=0}^{q-1} 2^j & & & & \\
2^{e_r} \arrow[ur] & & 2^{e_q} + \sum_{j=0}^{q-1} 2^j \arrow[ul, dotted] & & & \\
 & 2^{e_q} \arrow[ur] & & \sum_{j=0}^{q-1} 2^j \arrow[ul] & & \\
 & & 2^{q-1} \arrow[ur] & & 2^1 + 2^0 \arrow[ul, dotted] & \\
 & & & 2^1 \arrow[ur] & & 2^0 \arrow[ul]
\end{tikzcd}$$

(a) Nodes with list lengths

$$\begin{tikzcd}[column sep=tiny]
 & \sum_{i=q}^{r-1} 2^{e_i} + 2^q - 1 & & & & \\
\overline{\mathcal{B}}(2^{e_r}) \arrow[ur] & & 2^q - 1 \arrow[ul, dotted] & & & \\
 & \overline{\mathcal{B}}(2^{e_q}) \arrow[ur] & & 2^{q-1} - 1 \arrow[ul] & & \\
 & & \overline{\mathcal{B}}(2^{q-1}) \arrow[ur] & & 2^1 - 1 \arrow[ul, dotted] & \\
 & & & \overline{\mathcal{B}}(2^1) \arrow[ur] & & \overline{\mathcal{B}}(2^0) \arrow[ul]
\end{tikzcd}$$

(b) Nodes with best delays

FIGURE 89: Best delay mergings for n odd

balanced merges: the merge tree is built bottom-up in a uniform manner. FIGURE 79 on page 302 shows the sequence in which the unbalanced merges are performed. Abstractly, we could order them as we wish, so the question naturally arises: does this scheme lead to the fastest algorithm in the worst or average case? FIGURE 79 can be considered as an extreme case, because the unbalanced merges are computed from right to left. Another extreme case is the exact reverse order: from left to right, as shown in FIGURE 90 on page 329. Would you expect the worst delay to be worse than merging leftward? Same question about the best delay.

9. Prove $n\lfloor \lg n \rfloor + 2n - 2^{\lfloor \lg n \rfloor + 1} = n\lceil \lg n \rceil + n - 2^{\lceil \lg n \rceil}$.
10. Use theorem of Question 4 to derive a better lower bound for $\mathcal{A}_n^{\mathsf{sort}}$ and then $1 \leqslant n \leqslant 26 \Rightarrow \mathcal{A}_n^{\mathsf{i2wb}} < \mathcal{A}_n^{\mathsf{sort}}$.

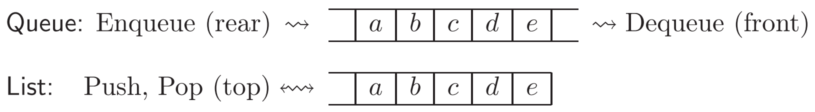
$$\begin{tikzcd}[column sep=tiny]
 & & & n & \\
 & & 2^{e_r} + \ldots + 2^{e_1} \arrow[ur] & & 2^{e_0} \arrow[ul] \\
 & 2^{e_r} + 2^{e_{r-1}} \arrow[ur, dotted] & & 2^{e_1} \arrow[ul] & \\
2^{e_r} \arrow[ur] & & 2^{e_{r-1}} \arrow[ul] & &
\end{tikzcd}$$

FIGURE 90: Unbalanced merges from left to right

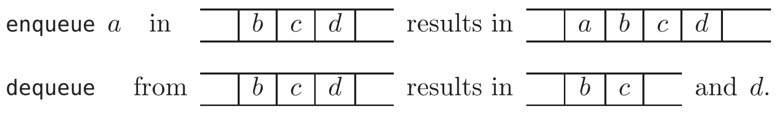
Chapter 14

Queues

A list can only be augmented with new items on only one of its ends, called the *top*. A *queue* is like a list where items are pushed on one end but popped *at the other end*. Adding an item to a queue is called *enqueuing*, whereas removing one is called *dequeuing*. The end of the queue used for input is called the *rear* and the other end is the *front*. Compare the following figures, where the arrows (⇝) indicate the way the items move:



Let `enqueue/2` be a function such that the call `enqueue(I,Q)` is rewritten into a queue whose last item in was `I` and remaining queue `Q`. Dually, let `dequeue/1` be a function such that the call `dequeue(Q)` is rewritten into the pair `{R,I}` where `I` is the first item out of queue `Q` (if there is none, `dequeue(Q)` is undefined) and `R` is the remaining queue.



One-list implementation. Let us first try to implement these functions on queues by means of the usual functions on one list. Let us rename `enqueue/2` into `enq1/2` and `dequeue/1` into `deq1/2` to remind that they rely on a single list. In this case, `enq1/2` is simply a push operation and `deq1(Q)` is a *pair* made of the last item in the list `Q` and the longest prefix of `Q` excluding it (that is, all the previous items in the same order). In other words, `deq1/1` evaluates in a pair whose first component is made of all the original items in the list except the last one. The second component is the last item in question. Consider the following:

$$
\begin{aligned}
\texttt{enq1(a,[])} &\twoheadrightarrow \texttt{[a]};\\
\texttt{enq1(a,[b,c])} &\twoheadrightarrow \texttt{[a,b,c]};\\
\texttt{deq1([])} &\not\twoheadrightarrow;\\
\texttt{deq1([a])} &\twoheadrightarrow \texttt{\{[],a\}};\\
\texttt{deq1([a,b,c,d,e])} &\twoheadrightarrow \texttt{\{[a,b,c,d],e\}}.
\end{aligned}
$$

We induce the following Erlang definitions:

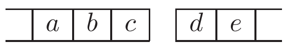
```
enq1(I,Q)   -> [I|Q].                         % Queue Q is a list
deq1(  [I]) -> {[],I};
deq1([I|Q]) -> {R,J}=deq1(Q), {[I|R],J}.
```

If the queue contains n items, then the delay $\mathcal{D}_n^{\mathsf{deq1}}$ is defined by the following recurrent equations:

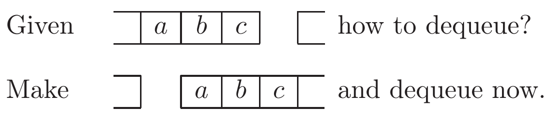
$$\mathcal{D}_1^{\mathsf{deq1}} = 1; \quad \mathcal{D}_n^{\mathsf{deq1}} = 1 + \mathcal{D}_{n-1}^{\mathsf{deq1}}, \text{ with } n > 0.$$

This is a recurrence extremely easy to solve: $\mathcal{D}_n^{\mathsf{deq1}} = n$, for $n \geqslant 0$. So this one-list implementation of queues provides constant time for enqueuing and linear time for dequeuing.

Two-list implementation. Let us now implement a more sophisticated approach with two lists, instead of one: one for enqueuing, called the *rear list*, and one for dequeuing, called the *front list*.

Enqueue (rear) ⇝  ⇝ Dequeue (front)

So enqueuing is pushing on the rear list and dequeuing is popping on the front list. In the latter case, if the front list is empty and the rear list is not, we swap the lists and reverse the (new) front. Pictorially, this is best summed up as



Let the pair `{In,Out}` denote the queue where `In` is the rear list and `Out` the front list. Enqueuing is as easy as with one list: just push the item on `In`:

```
enq2(E,{In,Out}) -> {[E|In],Out}.
```

Dequeuing requires some care when the front list is empty. This suggests an approach like this:

`deq2({In,     []}) ->` $\boxed{\qquad\qquad\qquad\qquad\qquad}$;
`deq2({In,[E|Out]}) ->` $\boxed{\qquad\qquad\qquad\qquad\qquad}$.

In the first clause, we must start over after moving the rear list to the front and reversing it (or reverse first and swap next, since both operations commute and are performed in one rewrite):

```
deq2({In,     []}) -> deq2({[],rev(In)});
deq2({In,[E|Out]}) -> □.

rev(L)            -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

In the second clause, we found the item expected to be dequeued: `E`. Therefore we draw

```
deq2({In,     []}) -> deq2({[],rev(In)});
deq2({In,[E|Out]}) -> {{In,Out},E}.
```

When testing a definition, it is wise to always envisage the “empty case,” whatever that means for the data structures at hand. In this particular case, the empty queue is uniquely implemented by the Erlang value `{[],[]}`. If we try to rewrite `deq2({[],[]})`, we realise that the program never terminates because the first clause always matches the empty queue. What should have been expected for the empty queue? Simply put, the function `deq2/1` should not be defined on the empty queue. Thus, let us change the pattern of the first clause in order to exclude any match for the empty queue:

```
deq2({In=[_|_],     []}) -> deq2({[],rev(In)});
deq2({      In,[E|Out]}) -> {{In,Out},E}.
```

Let $\mathcal{D}^{\mathsf{deq2}}_{p,q}$ be the delay for computing the call `deq2({`P`,`Q`})`, where p and q are, respectively, the lengths of the rear list P and the front list Q. From the definition, two recurrent equations follow:

$$\mathcal{D}^{\mathsf{deq2}}_{p,0} = 1 + \mathcal{D}^{\mathsf{rev}}_{p} + \mathcal{D}^{\mathsf{deq2}}_{0,p} = p + 3 + \mathcal{D}^{\mathsf{deq2}}_{0,p}, \text{ with } p > 0;$$

$$\mathcal{D}^{\mathsf{deq2}}_{p,q} = 1, \text{ with } q > 0.$$

The best case for dequeuing happens when the front list is not empty, that is, $q > 0$: the delay is then $\mathcal{B}^{\mathsf{deq2}}_{n} = 1$, where $n > 0$ is the total number of items in the queue. The worst case consists in having the front list empty, that is, $q = 0$. Then the delay is $\mathcal{W}^{\mathsf{deq2}}_{n} = n + 4$.

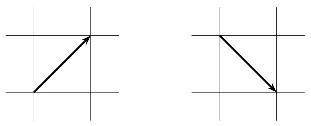
Comparison. We have $\mathcal{B}^{\mathsf{deq2}}_{n} \leqslant \mathcal{D}^{\mathsf{deq1}}_{n} < \mathcal{W}^{\mathsf{deq2}}_{n}$, therefore it is not possible to conclude for which values of n `deq1/1` is faster than `deq2/1`. A finer analysis is needed. Because the worst delay of `deq/2` depends on the number of items in the rear list, that is, on the number of previous enqueuings, it makes sense to study the delay of *a sequence of operations* (here, enqueuings and dequeuings) on a two-list queue, *in the* *worst case*. Up to now, we studied worst case scenarios of *one* function, taken in isolation, and we just found that this is sometimes insufficient. In general, it is indeed possible that the worst case for one specific operation leads to a good case for other subsequent operations; therefore, *adding the worst case delays of all operations taken in isolation may be much more than the worst delay of the series considered as a whole.* This kind of analysis is called *amortised* because it takes into account the fact that, in a series of operations on a data structure, some configurations lead to slow operations but the treatment of these create good configurations for others, so the high delay is amortised on a long run. Note that amortised analysis is not a statistic or probabilistic argument, as the randomness of a configuration has nothing to do with the phenomenon targeted by amortised analysis: we still consider the worst case, but of a sequence of operations, not a single one.

To contrast these two analyses, let us study the delay $\mathcal{D}_n$ of a series of n operations on a queue, consisting of enqueuings and dequeuings implemented, respectively, by `enq2/2` and `deq2/1`. First, let us consider the worst case of a single operation. We already know that the worst case is when the operation is a dequeuing and the front list is empty: $\mathcal{W}_i^{\mathsf{deq2}} = i + 4$, where i is the number of items in the rear list. This means that

$$\mathcal{D}_n \leqslant \sum_{i=1}^{n-1} \mathcal{W}_i^{\mathsf{deq2}} = \frac{1}{2}(n-1)(n+8) \sim \frac{1}{2}n^2.$$

There exists a real number α and an integer n_0 such that, for all $n > n_0$, $|(n-1)(n+8)/2| \leqslant \alpha n^2$. This is usually written concisely in the notation introduced by the mathematician Paul Bachmann as $\mathcal{D}_n \in \mathcal{O}(n^2)$. That is, the delay of n operations in the worst case is bounded from above by a quadratic polynomial as n gets large. Contrast this notation with Landau's notation in equation 13.59 on page 321, which is the set of functions $o(g(n))$ such that $f \in o(g(n)) \Leftrightarrow \lim_{n\rightarrow\infty} f(n)/g(n) = 0$.

Actually, the worst case above is too pessimistic and cannot even exist. Indeed, two important constraints were left aside. First, one cannot dequeue on an empty queue. As a consequence, at any time, the number of enqueuings since the beginning is always greater or equal than the number of dequeuings. In particular, the series of operations must start with at least one enqueuing. Second, when dequeuing on a queue whose front list is empty, all the items of the rear list are reversed into the front list, so they cannot be reversed again during the next dequeuing, whose delay will be 1. Moreover, $\mathcal{D}^{\mathsf{enq2}} \leqslant \mathcal{D}_n^{\mathsf{deq2}}$, so the first consequence is that the worst case for a series of n operations is when the number of dequeuings is maximum, that is, when it is $\lfloor n/2 \rfloor$.



(a) Enqueue (b) Dequeue

FIGURE 91: Graphical representations of operations on queues

Thus, let us distinguish an even number of operations and an odd number. If we denote e and d the number of enqueuings and, respectively, the number of dequeuings, we have $n = e + d$ and the two requisites for a worst case become $e = d$ or $e = d + 1$.

Dyck path ($e = d$). This refinement still does not take into account all the information we have, for instance, we do not use the fact that *at any time* the past number of enqueuings is greater or equal than the past number of dequeuings (we only assumed this to hold after n operations). This constraint can be visualised by representing an enqueuing by an opening parenthesis and a dequeuing by a closing parenthesis, so, for example, a valid series of operations when $e = d = 7$ could be `((()()(()))())`. All the prefixes of this string of parentheses are

```
( (( ((( ((() ((()( ((()() ((()()( ((()()(( ((()()(()
  ((()()(()) ((()()(())) ((()()(()))( ((()()(()))()
                ((()()(()))())
```

We can check that for each of them the number of closing parentheses never exceeds the number of opening parentheses. Another way to visualise this example is graphically, using a similar convention to the one found in section 8 on page 175. See FIGURE 91 on page 335. Then, `((()()(()))())` can be represented in FIGURE 92 on page 336 as a *Dyck path*, named in the honour of the logician Walther (von) Dyck. For a broken line to qualify as a Dyck path of *length* n, it has to start at the origin $(0, 0)$ and end at coordinates $(n, 0)$, that is, on the abscissa axis. In terms of a *Dyck language*, an enqueuing is called a *rise* (see FIGURE 91a on page 335) and a dequeuing is called a *fall*. A rise followed by a fall is called a *peak*. The ordinate of a point on a path is called its *height*. (The continued analogy is that of a mountain range.) For instance, in FIGURE 92 on page 336, there are four peaks. The number near each rise or fall is the delay incurred by the corresponding operation on the queue, that is, either an enqueuing or a dequeuing. The height is interpreted as the number of items in the queue at a given time. Time is the meaning of the abscissas.

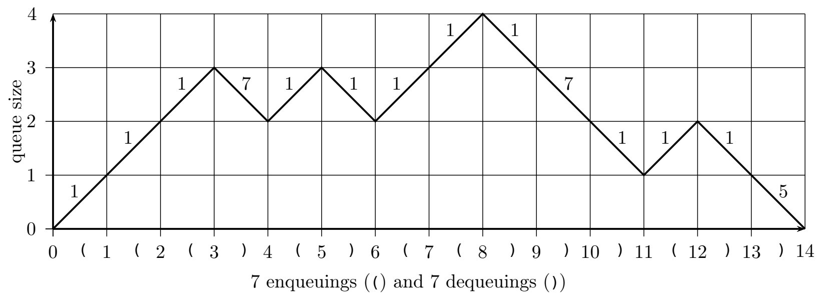


FIGURE 92: Dyck path modelling queue operations (delay 30)

When the number of enqueuings equals the number of dequeuings, that is, $e = d$, the graphical representation of the operations is a Dyck path of length $n = 2e = 2d$. In order to deduce the total delay in this case, we must find a decomposition of the path that is workable. By decomposition we either mean to identify patterns whose delays are easy to compute and which make up any Dyck path, or to associate any path to another path whose delay is the same but easy to find. Actually, we are going to do both. FIGURE 93 on the following page shows how the previous path can be mapped to an equivalent path only made of a series of isosceles triangles whose bases belong to the abscissa axis. Let us call them *mountains* and their series a *range*. The algorithm for the mapping is simple: after the first fall, if we are back to the abscissa axis, we have a mountain and we start again with the rest of the path. Otherwise, the next operation is a rise and we exchange it with the first fall after it. This brings us down by 1 and we can repeat the procedure until the bottom line is reached. We call this process *rescheduling* because it amounts, in operational terms, to reordering subsequences of

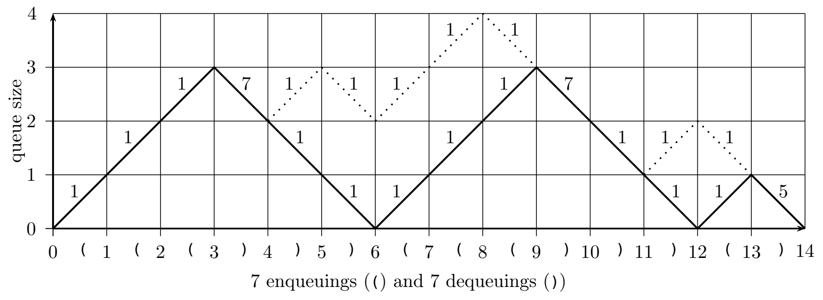


FIGURE 93: Dyck path equivalent to FIGURE 92 on the following page

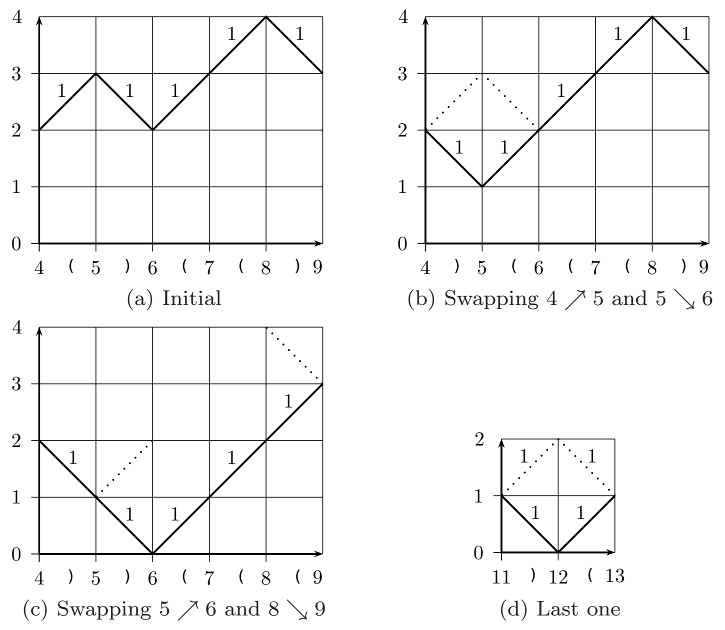


FIGURE 94: Rescheduling of FIGURE 92 on page 336

operations a posteriori. For instance, see in FIGURE 94 on page 337 the rescheduling of FIGURE 92 on the following page. Rescheduling is not an *injection*, that is, two different Dyck paths can be rescheduled to the same path. What makes FIGURE 94c on the next page, called a *low valley*, equivalent to FIGURE 94a on the facing page is that the delay is invariant because all operations have individual delays of 1. This is always the case because, on the one hand, enqueuings always have delay 1 and, on the other hand, the dequeuings involved in a rescheduling have delay 1 because they found the front list non-empty after a peak. We proved that all series of queue operations corresponding to a Dyck path are equivalent to a range, whose associated queue operations have the same delay as the original one. Therefore, the worst case can be found on ranges alone and it happens that their delays are easy to compute. Let us note $e_1, e_2, \ldots, e_k$ the series of rises; for example, in FIGURE 93, we have $e_1 = 3$, $e_2 = 3$ and $e_3 = 1$, meaning: "Three rises from the bottom line, then again and finally one rise." Of course, $e = e_1 + e_2 + \cdots + e_k$. Then the fall making up the ith peak incurs the delay $e_i + 4$ due to the front list being empty because we started the rise from the abscissa axis. The next $e_i - 1$ falls have all

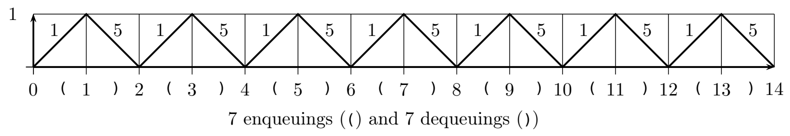


FIGURE 95: Worst case when $e = d = 7$ (delay 42)

delay 1, because the front list is not empty. For the ith mountain, the delay is thus $e_i + (e_i + 4) + (e_i - 1) = 3e_i + 3$. In total, the delay $\mathcal{D}_{e,k}$ is thus

$$\mathcal{D}_{e,k} = \sum_{i=1}^{k}(3e_i + 3) = 3(e + k).$$

We can check the example in FIGURE 93 on page 336, where $e = 7$ and $k = 3$ and find $\mathcal{D}_{e,k} = 3 \cdot (7 + 3) = 30$, which is correct. The worst case is obtained by maximising $\mathcal{D}_{e,k}$ for a given e, because k, not being a parameter of the problem, is free to vary. We have

$$\max_{1 \leqslant k \leqslant e} \mathcal{D}_{e,k} = \mathcal{D}_{e,e} = 3(e + e) = 3n = \mathcal{W}_{e,e}, \text{ with } n = 2e,$$

where $\mathcal{W}_{e,e}$ is the worst delay when there are e enqueuings and $d = e$ dequeuings. In other words, the worst case when $e = d = 7$ is the saw-toothed Dyck path shown in FIGURE 95 on the next page. Importantly, there are no other Dyck paths whose rescheduling lead to this worst case and the reason is that the reverse transformation from ranges to general Dyck paths works on dequeuings of delay 1 and the solution we found is the only one with no dequeuing equal to 1.

Dyck meander ($e = d + 1$). The other possibility for a worst case is that $e = d + 1$ and the graphical representation is then a *Dyck meander* whose extremity ends at ordinate $e - d = 1$. An example is given in FIGURE 96 on the following page, where the last operation is a dequeuing. The dotted line delineates the result of applying the rescheduling we used on Dyck paths. Here, the last operation becomes an enqueuing. Another possibility is shown in FIGURE 97 on page 339, where the last operation is left unchanged. The difference between the two examples lies in the fact that the original last dequeuing has, in the former case, a delay of 1 (thus is changed) and, in the latter case, a delay greater than 1 (thus is invariant). The range resulting from the rescheduling leads to peaks whose falls have a delay greater than 1. The third kind of Dyck meander to consider is one ending with an enqueuing, but because this enqueuing must start from the abscissa axis, this is the same situation as the result of rescheduling a meander ending with a dequeuing with delay 1 (see dotted line in FIGURE 96 again). Therefore, we are left to

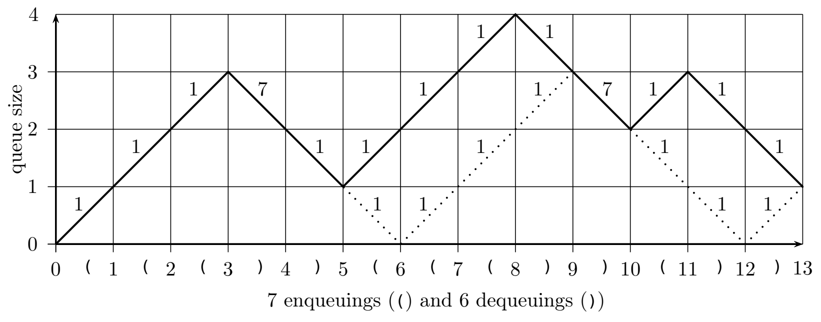


FIGURE 96: Dyck meander modelling queue operations (total delay 25)

compare the results of rescheduling meanders ending with a dequeuing, that is, we have two cases: either a range of $n-1$ operations ($e-1$ enqueuings and $d=e-1$ dequeuings) followed by an enqueuing or a range of $n-3$ operations ($e-2$ enqueuings and $d-1=e-2$ dequeuings) followed by two rises and one fall (totalling a delay of 8). In the former case, the worst case of the range is a saw-toothed Dyck path of delay $\mathcal{D}_{e-1,e-1} = 3((e-1)\cdot 2) = 3(n-1)$, because $n = e+d = 2e-1$. The worst delay of the whole meander is then simply $3(n-1)+1 = 3n-2$ and there are $k = e-1$ peaks. In the latter case, the worst case of the range is a saw-toothed Dyck path of delay $\mathcal{D}_{e-2,e-2} = 3(n-3)$, so the worst delay of the whole meander is $3(n-3)+8 = 3n-1$. This delay is slightly worse than the previous one, so we can conclude

$$\mathcal{W}_{e,e-1} = 3n-1, \text{ with } n = 2e-1.$$

Conclusion. As a conclusion, the delay $\mathcal{D}_n$ of a series of n queue operations, that is, enqueuings and dequeuings starting on an empty

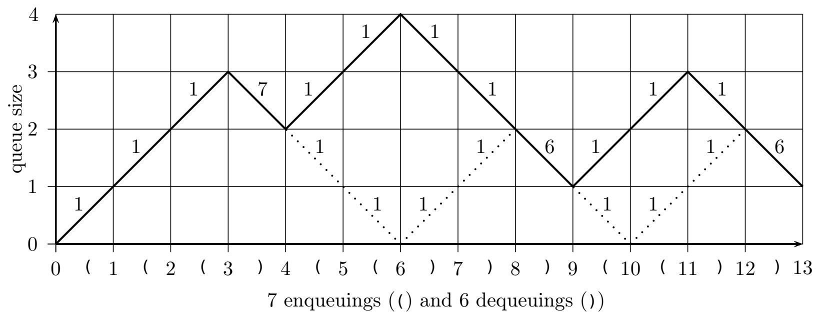


FIGURE 97: Dyck meander modelling queue operations (total delay 29)

queue, is bounded as follows:

$$n \leqslant \mathcal{D}_n \leqslant 3n.$$

The lower bound corresponds to the case where all the operations are enqueuings, the upper bound occurs when there are e enqueuings and e dequeuings, with $n = 2e$, in a saw-toothed pattern. Because these bounds are tight, that is, they correspond to actual configurations of the input, it would be less precise to state that $\mathcal{D}_n \in \mathcal{O}(n)$, where the multiplicative constant is hidden and unknown. We proved that, despite some dequeuings having a delay linear in the size of the queue in the worst case, *a series of enqueuings and dequeuings has a linear delay in terms of the number of operations in the worst case,* because individual worst cases lead to favourable dequeuings later. If we average the delay over the number of operations, we obtain the *amortised delay* of one operation, $\mathcal{D}_n/n$, which lies between 1 and 3.

Exercises. [See answers page 378.]

1. Write a definition for a function `head/1` such that the call `head(Q)`, where `Q` is a queue implemented with two lists, is rewritten into the first item to be dequeued. For example,

   `head({[4,3],[1,2]})` $\twoheadrightarrow$ `1`.

   Make sure that the delay of `head/1` is constant. (*Hint.* Revise `enq2/2` and `deq2/1` so that they enforce the invariant property that the front list is empty only if the rear list is empty. In other words, suppose that this is true when the heads match and make sure it is still true after the body is computed.)
2. A *double-ended queue* is a queue where enqueuing and dequeuing are allowed on both ends. Use two lists for the implementation and write enqueuing and dequeuing on both ends efficiently. (*Hint.* Extend the invariant of the previous exercise so that none of the two lists is empty if the double-ended queue contains at least two items and when a list becomes empty, split the other list in half and reverse one of the halves.)

Chapter 15

Factor Matching

In chapter 4 on page 75, we introduced the basic algorithm of sequential search which consists in searching step by step for the occurrence of a given item into a given list. We can think of generalising it by searching for *a series* of items into a list. By “series” we mean a shorter list whose items may occur consecutively in the larger list. For example, we say that the list `[c,d]` occurs in the list `[a,b,c,d,e]` or, more precisely, that the former is a *factor* of the latter. Let us call this very common problem *factor matching*. For instance, when using text editors, the need to search for a word in a text arises frequently. In bioinformatics, it is fairly common to search a genome for a given sequence of base pairs. Of course, it is interesting to know where the factor is located and this information will be represented by an integer, called *index*, corresponding to the position at which the head of the factor is first found, given that the head of a list has index 0. We are not concerned here with finding all the factors, only the first. For instance, if we look for the list `[b,r]` in `[a,b,r,a,c,a,d,a,b,r,a]`, we expect as a result the index 1 only. Note that the lists can hold integers or any other kind of data. For the sake of clarity, we refer to parts of lists by means of the indexes delimiting them. For example, if t is `[a,b,r,a,c,a]`, then we write $t[2] =$ `r` and $t[4] =$ `c`. The list $w =$ `[r,a,c]` is a factor of t at index 2 because $t[2..4] = w$, which is a shortcut for the equalities $t[2] = w[0]$, $t[3] = w[1]$ and $t[4] = w[2]$. Now we can capture the essence of factor matching in FIGURE 98 on page 342, where t is the list in which x is sought and $a_i := t[i]$.

1. First, let us suppose that an occurrence of a_0 has been found at index $j - i$ in t, that is, $x[0] = t[j - i]$.
2. Next, it is checked that a_1, a_2, etc. till a_{i-1} occur successively in t, that is, $x[0..i - 1]$ is a factor of t starting at index $j - i$. The failure happens at index j in t and index i in x, that is, $t[j] \neq x[i]$.

To express this, we arbitrarily chose to set $t[j] = \texttt{b}$ and $x[i] = \texttt{a}$.

3. If $t[j-i+1]$ is undefined, then x is not a factor of t. Otherwise, we compare $t[j-i+1..]$ to $x[0]$: if they match, we resume step 2, otherwise step 3, in both cases with $j-i+1$ instead of $j-i$.

We write $t[j-i+1..]$ to mean the sub-list of t starting at index $j-i+1$. This kind of sub-list is called a *suffix*, for example, `[1,0,1]` is a suffix of `[0,1,1,1,0,1]`. Suffix is just another name for sub-list, just as stack is another name for list, and this is why, incidentally, it would be incorrect to claim that `[c,d]` is a sub-list of `[a,b,c,d,e]`. When a factor starts at index 0, it is called a *prefix*, for example, `[0,1,1]` is a prefix of `[0,1,1,1,0,1]`. A prefix which is not identical to the whole list is said to be a *proper prefix*. The same is said for *proper suffixes*. A list can be both a prefix and a suffix of another, for example, `[0,1]` is both a prefix and a suffix of `[0,1,1,1,0,1]`. If a proper prefix is a suffix, then it is called a *side*. Step 3 above can be thought of as if x is slided along t of one index before it is checked whether it is a prefix of the current suffix of t. The algorithm we just described can be implemented in Erlang as

```
find(Word,Text) -α-> find(Word,Text,0).        % First index is 0

find(_,     [],_) -β-> absent;                          % Failure
find(X,T=[_|U],J) -γ-> case prefix(X,T) of
                         yes -δ-> {factor,J};           % Success
                         no  -ε-> find(X,U,J+1)         % Sliding
                       end.

prefix(   [],     _) -ζ-> yes;                          % Success
prefix([A|Y],[A|T]) -η-> prefix(Y,T);                     % Match
prefix(    _,     _) -θ-> no.                          % Mismatch
```

Notice how function `find/3` returns either the atom `absent` if `Word` is not in `Text`, or the pair `{factor,N}`, where `N` is the index in `Text` at which `Word` occurs. This implementation can be considered as the

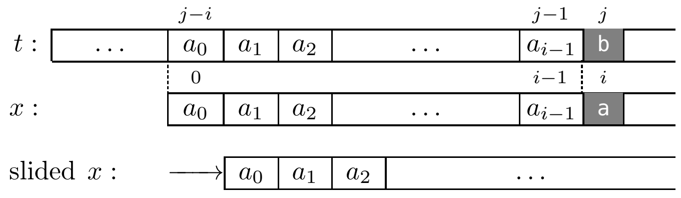


FIGURE 98: Naive factor matching (failure in grey, then sliding)

interleaving of a sequential search and a prefix check. Actually, we can make the code shorter by tightening further this relationship because clause ζ is always immediately followed by clause δ and clause θ is followed by ϵ. Therefore, we could get rid of the `case` construct and let `prefix/2` continue with what is left to be done, at the cost of passing more arguments to `prefix/2`. The result is definitions in tail form.

```
find(Word,Text) -> find(Word,Text,0).        % First index is 0

find(_,     [],_) -> absent;                           % Failure
find(X,T=[_|U],J) -> prefix(X,T,X,U,J).

prefix(   [],    _,_,_,J) -> {factor,J};               % Success
prefix([A|Y],[A|T],X,U,J) -> prefix(Y,T,X,U,J);          % Match
prefix(    _,    _,X,U,J) -> find(X,U,J+1).            % Sliding
```

In our delay model we do not count the time to execute a `case` construct, so these two versions have the same delay. On the other hand, a `case` must take some real time. Also, passing more arguments should increase the delay, but we do not take this into account neither. So, following our assumptions, the differences between the two pieces of code are merely stylistic, which does not mean that they do not matter. For instance, the mind finds it more difficult to remember many parameters when writing the function bodies. Also, the first version, although longer, makes evident the discarded information: when returning from a call to `prefix/2`, only `yes` or `no` matter. In the former case, it is fine, but we may wonder later if we could not use the partial match of `prefix/2` to speed up `find/3`. In the mean time, we can shorten the code above by noting that we can expand the definition of `find/3` call `find(X,U,J+1)` and get rid of `find/3` altogether:

```
find(Word,Text) -> prefix(Word,Text,Word,Text,0).

prefix(   [],    _,_,    _,J) -> {factor,J};
prefix(    _,   [],_,    _,_) -> absent;
prefix([A|Y],[A|T],X,    U,J) -> prefix(Y,T,X,U,J);
prefix(    _,    _,X,[_|U],J) -> prefix(X,U,X,U,J+1).
```

Notice that we actually improved the delay if we reach the end of the text whilst the word is not exhausted yet: in other words, when the word does not occur in the text.

In the following discussion about delays, remember that the delay attached to arrows clauses δ and ϵ is 0 because we assume that they are compiled differently and more efficiently than a function call. (Formally,

$|\delta| = |\epsilon| = 0$.) Also, we shall call *word* a list when it is written using the lighter notation consisting in stripping the opening and closing square brackets and the commas of the Erlang lists, so, for instance, abc is the word corresponding to the list [a,b,c] (note also the different typeface to avoid confusion between the atom abc and the word abc). This notation proves to be very useful when the items are atoms or integers, which we may call *letters*. In the current context of factor matching, it is convenient to distinguish between the word whose occurrence is being searched and the one in which it is looked for. Following the analogy of text editing, we shall call the latter *text*. Let m be the length of the word x and n the length of the text t.

Best Delay. The best case happens when the word is a prefix of the text. The execution trace corresponding to the best case is $\alpha\gamma\eta^m\zeta\delta$, so the best delay is $\mathcal{B}^{\mathsf{find}}_{m,n} = m + 3$ (δ is not counted in).

Worst Delay. Let us make three cases, depending on the relative lengths of the word (m) and the text (n).

1. *We have $m > n$.* The worst case occurs then when t is a prefix of x, so prefix/2 always fails because the text is exhausted. Consider $t = \mathsf{a}^n$ and $x = \mathsf{a}^m$. The corresponding execution trace has length
$$\left|\alpha \cdot \prod_{i=0}^{n-1}(\gamma\eta^{n-i}\theta\epsilon) \cdot \beta\right| = 1 + \sum_{i=1}^{n}(1 + i + 1) + 1 = \frac{1}{2}n^2 + \frac{5}{2}n + 2.$$
2. *We have $m = n$.* In this case, the worst case is when prefix/2 fails on the last letter of the arguments and subsequent calls fail because the text is exhausted (so the proper suffixes of t are completely traversed). This means that $x[0..m-1]$ is the longest prefix of t which is also a suffix. For example, consider $t = \mathsf{a}^n$ and $x = \mathsf{a}^{m-1}\mathsf{b}$. The corresponding execution trace has the length
$$\left|\alpha \cdot (\gamma\eta^{n-1}\theta\epsilon) \cdot \prod_{i=1}^{n-1}(\gamma\eta^{n-i}\theta\epsilon) \cdot \beta\right| = \frac{1}{2}n^2 + \frac{5}{2}n + 1.$$
We already see that the case 1 ($m > n$) leads to a slightly worse worst delay. One more case remains.
3. *We have $m < n$.* Two subcases can be made, depending whether the word occurs or not in the text.
   (a) *Let us assume that there is an occurrence of x in t.* The worst case is when x is a suffix of t, therefore the word is matched unsuccessfully at all the indexes in the text from 0 to $n-m-1$, both included, and successfully starting at index $n-m$ in t. The delay for a word mismatch is maximum when the failure

happens at the last letter of the word. An example is $t = \mathsf{a}^{n-1}\mathsf{b}$ and $x = \mathsf{a}^{m-1}\mathsf{b}$, where a^i denotes the list of length i containing only a and $u \cdot v$ or, in short, uv, denotes the list resulting from appending list v to list u. The execution trace corresponding to this case is thus $\alpha(\gamma\eta^{m-1}\theta\epsilon)^{n-m}(\gamma\eta^{m}\zeta\delta)$. The total delay is thus $1+(1+(m-1)+1)(n-m)+(1+m+1)$, that is, $(m+1)n - m^2 + 3$.

(b) *Let us suppose that there is no occurrence of x in t.* The word x is not the prefix of any suffix of t. More precisely, from indexes 0 to $n-m$ in t, the mismatch happens at the last letter of x. Indexes from $n-m+1$ to $n-1$ in t are tried unsuccessfully with failure occurring at indexes $m-1$ to 0 in x. The final failure is then due to the text being exhausted (see clause β above). An example of match failure in the worst case is $t = \mathsf{a}^n$ and $x = \mathsf{a}^{m-1}\mathsf{b}$. The corresponding execution trace is hence

$$\alpha(\gamma\eta^{m-1}\theta\epsilon)^{n-m+1} \cdot \prod_{i=1}^{m}(\gamma\eta^{m-i}\theta\epsilon) \cdot \beta,$$

whose length is

$$(m+1)n - \frac{1}{2}m^2 + \frac{3}{2}m + 1.$$

This delay is strictly greater than in case 3a, that is, $(m+1)n - m^2 + 3$, for any $m > 1$. They are equal if, and only if, $m = 1$ (the other root is $m = -4$, which makes no sense). Since here $m < n$, we derive a simple upper bound

$$(m+1)n - \frac{1}{2}m^2 + \frac{3}{2}m + 1 < \frac{1}{2}n^2 + \frac{5}{2}n + 1,$$

therefore the worst case is case 1, that is, when $m > n$, hence

$$\mathcal{W}^{\mathsf{find}}_{m,n} = \frac{1}{2}n^2 + \frac{5}{2}n + 2.$$

Average Delay. Let us suppose that $m < n$ and that the letters of x and t are chosen from the same finite set, called *alphabet*, whose cardinal is $\breve{a}$. More precisely, we shall assume that the letters are randomly picked in a uniform and independent manner, that is to say, all letters are equally likely to be chosen and the selection of one letter is independent of the other choices. This implies that letters may be repeated. Unfortunately, this also entails that these assumptions are not realistic if we think of editing a text in English or processing a DNA strand, but it is easily amenable to analysis and therefore has a real theoretical value, in particular in order to compare the naive search with improved versions of it. Let us start by investigating the behaviour of `prefix/2`.

First, let us consider an erroneous reckoning. Because the delay of `prefix/2` is $\mathcal{D}^{\mathsf{prefix}}_{m,n,k} = k+1$, where k is the index where a mismatch happens, assuming that $k = n$ means that no mismatch happened. Then the average delay is the average of the delays for all possible k, ranging from 0 to $n-1$ if we are interested in the case of a failure:

$$\mathcal{A}^{\mathsf{prefix}}_{m,n} = \frac{1}{n}\sum_{k=0}^{n-1}\mathcal{D}^{\mathsf{prefix}}_{m,n,k}.$$

This is what we did to obtain equation (4.8) on page 88 but it is here incorrect. The reason is that the variable k does not represent a randomly uniform choice because it represents a *series* of letter choices, as it is the end of a prefix.

Here is the correct approach. The informal and intuitive use of probability theory above can be avoided and the reckoning made formal in the following manner. Let us suppose that x is given. We need to compute and add the delays of the calls to `prefix/2` for all the possible choices of $t[0..m-1]$, which is matched against x, and divide the total by the number of choices. In FIGURE 99 on page 346 we represent all possible choices as a tree when the alphabet is $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and $m = 2$. The root is at level 0, the children of the root make up level 1 etc. Level $k > 0$ represents the set of all prefixes $x[0..k-1]$, whose number is thus $\breve{a}^k$. For example, level 1 represents the $\breve{a}$ possible values for $t[0]$, whereas level 2 stands for the $\breve{a}^2$ possible words $t[0..1]$. Graphically, a letter annotates each edge in the tree and a prefix of length k is a path from the root to a node in level k or, equivalently, a path from the root of length k. For instance, the word $x = \mathsf{ba}$ is distinguished in the figure with black nodes. What is the average delay of `prefix/2`? The total number of words of length 2 is the number of leaves in the tree, here 3^2. In general, this would be $\breve{a}^m$. Now, let us compute the sum of the delays of `prefix(`x`,`t`)`, where t ranges over all the texts in the

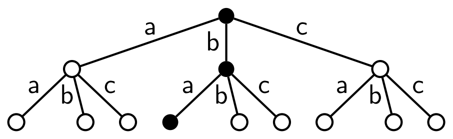


FIGURE 99: All words of length 2 over the alphabet $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$

tree T. Then the average delay is formally defined as

$$\mathcal{A}_{m,n}^{\texttt{prefix}} := \frac{1}{\breve{a}^m}\sum_{t\in T}[\![\texttt{prefix(}x\texttt{,}t[0..m-1]\texttt{)}]\!],$$

Notice that this value is independent of n, because the hypotheses we made imply that $\mathcal{A}_{m,n}^{\texttt{prefix}}$ does not depend on the current suffix of the original text. We shall come back on this when finally expressing $\mathcal{A}_{m,n}^{\texttt{find}}$. In the meantime, considering the instance in FIGURE 99 on page 346, this expression becomes, enumerating the texts aa, ab, ac, ba, bb, bc, ca, cb, cc:

$$\mathcal{A}_{2,n}^{\texttt{prefix}} = \frac{1}{9}([\![\texttt{prefix([b,a],[a,a])}]\!] + \cdots + [\![\texttt{prefix([b,a],[c,c])}]\!]).$$

These texts are paths of two kinds: either paths corresponding to a mismatch, that is, a result `no`, or the only path corresponding to a success, that is, a result `yes`, which is distinguished in bold in FIGURE 99 on page 346. The delay of the latter is $[\![\texttt{prefix([b,a],[b,a])}]\!]$, which is the length of the execution trace $\eta^2\zeta$, that is, 3. In general, this would be $m+1$. The delays in case of mismatch are the other paths in the tree. Let us compute them top-down, that is, starting from the root and ending at the leaves. First, we see that the subtrees after following the edges a and c have the exact same shape, so they denote the same number of texts. There are 2 of these subtrees and, in general $\breve{a}-1$. The number of texts in each of them is 3 and, in general, $\breve{a}^{m-1}$. The delay of `prefix/2` for all these texts is the same: $|\theta| = 1$. The same observation holds after following the edge corresponding to the second letter of ba: $\breve{a}-1$ subtrees containing each $\breve{a}^{m-2}$ texts, each of them leading to a delay for `prefix/2` equal to $|\eta\theta| = 2$. In general,

$$\begin{aligned}
\breve{a}^m \cdot \mathcal{A}_{m,n}^{\texttt{prefix}} &= (\breve{a}-1)\sum_{p=0}^{m-1}(m-p)\breve{a}^p + (m+1)\\
&= \sum_{p=0}^{m-1}(m-p)\breve{a}^{p+1} - \sum_{p=0}^{m-1}(m-p)\breve{a}^p + m + 1\\
&= \sum_{p=1}^{m}(m-p+1)\breve{a}^p - \sum_{p=0}^{m-1}(m-p)\breve{a}^p + m + 1 = \sum_{p=0}^{m}\breve{a}^p.
\end{aligned}$$

Supposing that the alphabet contains at least two letters, that is, $\breve{a} > 1$,

$$\mathcal{A}_{m,n}^{\texttt{prefix}} = \sum_{p=0}^{m}\frac{1}{\breve{a}^p} = \frac{1}{\breve{a}-1}\left(\breve{a} - \frac{1}{\breve{a}^m}\right) < \frac{\breve{a}}{\breve{a}-1} \leqslant 2.$$

We can derive $\mathcal{A}^{\texttt{find}}_{m,n}$ if we consider that clause α is always called once and, in the worst case,

- the word x is tried at every index of t from 0 to $m - n$, that is, `prefix/2` is called $m - n + 1$ times;
- all the proper prefixes of x are tried at indexes in t from indexes $n - m + 1$ to $n - 1$;
- clause β is called once.

Reusing the upper bound $\mathcal{A}^{\texttt{prefix}}_{m,n} \leqslant 2$, this analysis entails

$$\begin{aligned}\mathcal{A}^{\texttt{find}}_{m,n} &= 1 + (n - m + 1)\mathcal{A}^{\texttt{prefix}}_{m,n} + \sum_{k=1}^{m-1} \mathcal{A}^{\texttt{prefix}}_{k,n} + 1 \\ &\leqslant 1 + 2(n - m + 1) + 2(m - 1) + 1 = 2n + 2.\end{aligned}$$

The naive factor matching algorithm is hence quite efficient in average. (Notice that, although writing $\mathcal{A}^{\texttt{find}}_{m,n} \in \mathcal{O}(n)$ is correct, it is less informative than our result because we give explicitly the coefficients.)

Morris-Pratt Algorithm. The slowness of the naive algorithm in the worst case is due to the fact that, in case of mismatch in `prefix/2`, it starts again to compare the first letters of x *without using the information of the partial success of the previous attempt.* Indeed, if the mismatch occurred at index i in x and index j in t, as featured in FIGURE 98 on the following page, we know

$$x[0..i-1] = t[j-i..j-1], \qquad x[i] \neq t[j],$$

but this information is forgotten. Instead, any subsequent search in the factor $t[j-i+1..j-1]$ could reuse the information that it is a factor of x: since the previous partial success tells us that $t[j-i+1..j-1] = x[1..i-1]$, it compares $x[0..i-2]$ to $x[1..i-1]$, that is, *x is compared to a part of itself.* If we know an index k such that $x[0..k-1] = x[i-k..i-1]$, that is, $x[0..k-1]$ is a *side* of $x[0..i-1]$, then we can resume comparing $t[j..]$ with $x[k..]$, so, the greater k, the more comparisons we skip.

The side of a non-empty list x is a proper prefix of x which is also a suffix, or, equivalently, a proper suffix which is also a prefix. For example, the *word* abacaba has the three sides ε, a and aba. Let us note $\mathcal{S}(x)$ the longest side of a non-empty word x. We shall call it *maximum side* of x. For instance, in FIGURE 100 on page 349 is the table of the maximum sides for the prefixes y of the word $x =$ abacabac. "y is a prefix of x" is noted $y \preccurlyeq x$ and "y is a proper prefix of x" is noted $y \prec x$. Each column corresponds to a prefix y of x, not just to a *letter*, that is, an item of the corresponding list. So, at index 6, we should read $\mathcal{S}(\underline{\text{aba}}\text{c}\underline{\text{aba}}) = \text{aba}$, at index 3, $\mathcal{S}(\text{abac}) = \varepsilon$, where ε is the

$y \preccurlyeq x$	a	b	a	c	a	b	a	c
$\mathcal{S}(y)$	ε	ε	a	ε	a	ab	aba	abac

FIGURE 100: Maximum sides for the prefixes y of the word $x =$ abacabac

empty word, that is, the empty list, because abac $= \underline{\varepsilon}\mathsf{abac}\underline{\varepsilon}$. Also, note that maximum sides can overlap, for example, aaaa $= \underline{\mathsf{aaa}}\mathsf{a} = \mathsf{a}\underline{\mathsf{aaa}}$, so $\mathcal{S}(\mathsf{aaaa}) = \mathsf{aaa}$.

The optimisation brought by Morris and Pratt to the naive search is depicted in FIGURE 101 on page 349. We see how, in case of a mismatch, x is slided of $i - k$ indexes, where k is the length of the maximum side of the longest prefix of x such that $x[i] \neq t[j]$. Therefore, it is a mistake to think that the greater the slide, the more comparisons are skipped. In fact, the opposite is true: the greater the side is, the more comparisons are saved and, in the best case, the slide is 1 and the maximum side length is $i - 1$. The letters of t which have been matched successfully against a letter in x will not be compared again, contrary to the naive approach we described earlier; in other words, the indexes in t are considered increasingly. Also, notice that, for the sake of simplicity, the figure shows non-overlapping sides, which may not be accurate in general (think again about aaaa). Consider the example in FIGURE 102 on page 350 where, in the end, x is not found to be a factor of t. The cells in grey correspond to mismatches.

Let us wonder how to compute the maximum side of a word. It is clear that $\mathcal{S}(a) = \varepsilon$, for all letter a. In case the word contains more than one letter, that is, it has the shape $a \cdot y$, where y is a word, we want to know $\mathcal{S}(a \cdot y)$. For example, this value is found to be $a \cdot \mathcal{S}^3(y)$ in FIGURE 103 on page 350. The idea is, recursively, to consider $a \cdot \mathcal{S}(y)$. If $a \cdot \mathcal{S}(y)$ is a suffix of y, then $\mathcal{S}(ay) = a \cdot \mathcal{S}(y)$. Otherwise, we must consider the maximum side of the maximum side of y, that is, $a \cdot \mathcal{S}^2(y)$

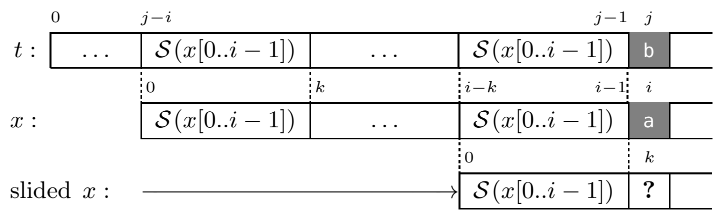


FIGURE 101: Morris-Pratt algorithm (failure in grey, then sliding)

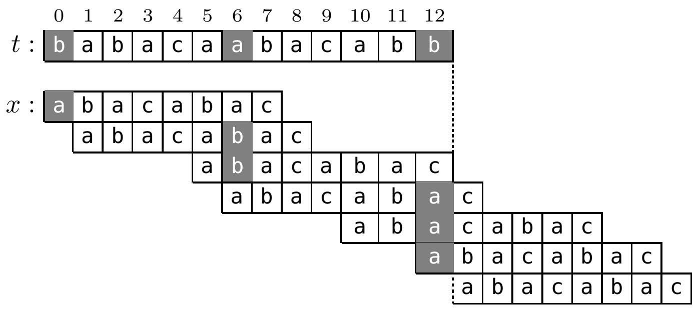


FIGURE 102: Morris-Pratt algorithm at work (no match found)

instead of $a \cdot \mathcal{S}(y)$, and iterate this process until $a \cdot \mathcal{S}^q(y)$ is a suffix of y, or else $\mathcal{S}^q(y)$ is ε. This can be formally summarised as follows. For all words $x \neq \varepsilon$ and all letters a:

$$\mathcal{S}(a) := \varepsilon;$$

$$\mathcal{S}(a \cdot y) := \begin{cases} a \cdot \mathcal{S}(y), & \text{if } a \cdot \mathcal{S}(y) \text{ is a suffix of } y; \\ \mathcal{S}(a \cdot \mathcal{S}(y)), & \text{otherwise.} \end{cases} \tag{15.60}$$

Consider the following examples where y and $\mathcal{S}(y)$ are given:

$$y = \mathsf{abaabb}, \quad \mathcal{S}(y) = \varepsilon, \quad \mathcal{S}(\mathsf{a} \cdot y) = \mathcal{S}(\mathsf{a} \cdot \mathcal{S}(y)) = \mathcal{S}(\mathsf{a}) = \varepsilon;$$
$$y = \mathsf{baaaba}, \quad \mathcal{S}(y) = \mathsf{ba}, \quad \mathcal{S}(\mathsf{a} \cdot y) = \mathsf{a} \cdot \mathcal{S}(y) = \mathsf{aba};$$
$$y = \mathsf{baabba}, \quad \mathcal{S}(y) = \mathsf{ba}, \quad \mathcal{S}(\mathsf{a} \cdot y) = \mathcal{S}(\mathsf{a} \cdot \mathcal{S}(y)) = \mathsf{a} \cdot \mathcal{S}^2(y) = \mathsf{a}.$$

This recursive definition is straightforwardly translated in Erlang as

```
side(  [_]) -> [];
side([A|Y]) -> Z = [A|side(Y)], case suffix(Z,Y) of
                                  yes -> Z;
                                  no  -> side(Z)
                                end.
```

Remains to define the function `suffix/2`. We already have a definition

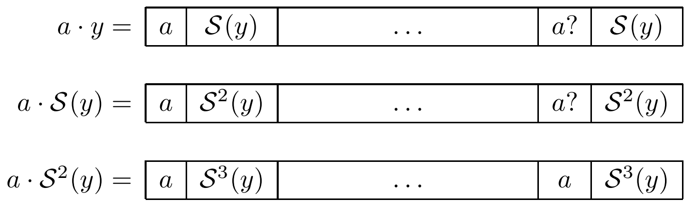

$a \cdot y =$
a | $\mathcal{S}(y)$ | $\ldots$ | $a?$ | $\mathcal{S}(y)$
$a \cdot \mathcal{S}(y) =$
a | $\mathcal{S}^2(y)$ | $\ldots$ | $a?$ | $\mathcal{S}^2(y)$
$a \cdot \mathcal{S}^2(y) =$
a | $\mathcal{S}^3(y)$ | $\ldots$ | a | $\mathcal{S}^3(y)$

FIGURE 103: $\mathcal{S}(a \cdot y) = \mathcal{S}(a \cdot \mathcal{S}(y)) = \mathcal{S}(a \cdot \mathcal{S}^2(y)) = a \cdot \mathcal{S}^3(y)$.

for `prefix/2`, would it be possible to reuse it? A sheet of paper and a pencil are enough to see that

$$\texttt{suffix}(U,V) \equiv \texttt{prefix}(\texttt{rev}(U),\texttt{rev}(V)),$$

where `rev/1` is the function reversing a list, efficiently defined as

```
rev(L)              -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Therefore

```
suffix(U,V) -> prefix(rev(U),rev(V)).
```

If we look back again at FIGURE 101 on the next page, we see that we are not really interested in knowing the maximum side of $x[0..i-1]$ but, instead, its length k, because the purpose is to discard the first k letters of x before resuming the comparisons. By means of a simple modification of `prefix/2`, it is possible to compute i:

```
prefix(X,T)            -> prefix(X,T,0).
prefix(   [],    _,_) -> yes;              % i useless if match
prefix([A|Y],[A|T],I) -> prefix(Y,T,I+1);
prefix(    _,    _,I) -> {no,I}.                        % I is i
```

Since it is easy to compute i, we should think of a direct way to compute the length of $\mathcal{S}(x[0..i-1])$ directly in terms of i. We could resume the derivation from the definition (15.60) on page 350, but it is actually going to be easier to work on a slightly different, equivalent definition based on the prefixes, not the suffixes. We used suffixes because we had in mind an implementation of a function matching lists, so having $\mathcal{S}(ay)$ made sense as `side([A|Y])`, whereas $\mathcal{S}(ya)$ would have not. Here, we do not have the concern of producing Erlang code which computes the side, so we should stick to the usage of the maximum side in FIGURE 101: we are interested to check the letter *after* the side, not before, so it makes more sense to wonder about $\mathcal{S}(ya)$, where $y := x[0..i-1]$. Moreover, this allows us to work solely with the concept of prefix instead of having to strive with both prefixes and suffixes (for which we do not have a notation, by the way). An alternate definition for $\mathcal{S}$ is: for any word $y \neq \varepsilon$ and any letter a,

$$\mathcal{S}(a) := \varepsilon;$$

$$\mathcal{S}(y \cdot a) := \begin{cases} \mathcal{S}(y) \cdot a, & \text{if } \mathcal{S}(y) \cdot a \preccurlyeq y; \\ \mathcal{S}(\mathcal{S}(y) \cdot a), & \text{otherwise.} \end{cases} \tag{15.61}$$

The rationale for it can be seen in FIGURE 104 on page 352, which is the dual of FIGURE 103 on the next page. Basically, instead of checking

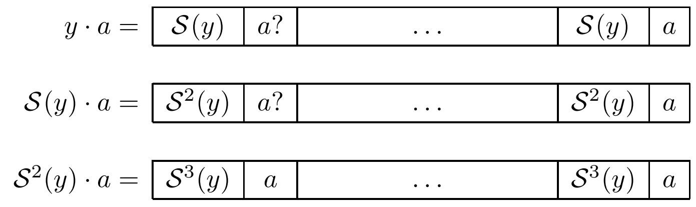


FIGURE 104: $\mathcal{S}(y \cdot a) = \mathcal{S}(\mathcal{S}(y) \cdot a) = \mathcal{S}(\mathcal{S}^2(y) \cdot a) = \mathcal{S}^3(y) \cdot a$.

the letter before the suffix, we check the letter after the prefix. Let us note $|y|$ the length of a word y. For a given word x, let us define a function μ_x on all its prefixes as

$$\mu_x(|y|) := |\mathcal{S}(y)|, \text{ for all } x \text{ and } y \neq \varepsilon \text{ such that } y \preccurlyeq x. \tag{15.62}$$

For reasons which will be clear latter, this function is called the *failure function* of x. An equivalent definition is

$$\mu_x(i) = |\mathcal{S}(x[0..i-1])|, \text{ for all } x \text{ and } i \text{ such that } 0 < i \leqslant |x|.$$

For example, FIGURE 105 on the following page shows the table of the maximum sides for the prefixes of the word $x = \mathsf{abacabac}$, this time with their lengths only. In FIGURE 101 on page 349, the length of the maximum side is k, so $k = \mu_x(i)$ and $x[\mu_x(i)]$ is the first letter to be compared with $t[j]$ after the slide. Also, the figure assumes that $i > 0$, so the side in question is defined. What if the mismatch happens between $t[j]$ and $x[0]$? Then this is the same as if a failure happened with the naive algorithm and x is slided of one index, so $x[0]$ is compared again but with $t[j+1]$. But this does *not* mean that we should extend μ such that $\mu_x(0) = 0$, because this equation denotes a comparison between $x[0]$ and $t[j]$ again, so we would be looping. In order to express the small step, we set

$$\mu_x(0) = -1. \tag{15.63}$$

In other words, $k = -1$, so the next index to the right is 0, which is what we wanted for the start of x after the slide. Of course, in the implementation, we will have to check whether $\mu_x(i) = -1$ before taking

x	a	b	a	c	a	b	a	c	
i	0	1	2	3	4	5	6	7	8
$\mu_x(i)$	□	0	0	1	0	1	2	3	4

FIGURE 105: Morris-Pratt failure function of abacabac

$x[\mu_x(i)]$. Let us continue on this track and reach a definition of μ which relies only on numerical values, without computing the maximum sides. The equations (15.60) on the facing page defining the maximum side can be unfolded as follows:

$$\begin{aligned}
\mathcal{S}(ya) &= \mathcal{S}(\mathcal{S}(y)\cdot a), & \mathcal{S}(y)\cdot a &\not\preccurlyeq y,\\
\mathcal{S}(\mathcal{S}(y)\cdot a) &= \mathcal{S}(\mathcal{S}^2(y)\cdot a), & \mathcal{S}^2(y)\cdot a &\not\preccurlyeq \mathcal{S}(y),\\
&\vdots\\
\mathcal{S}(\mathcal{S}^{p-2}(y)\cdot a) &= \mathcal{S}(\mathcal{S}^{p-1}(y)\cdot a), & \mathcal{S}^{p-1}(y)\cdot a &\not\preccurlyeq \mathcal{S}^{p-2}(y),
\end{aligned}$$

and $\varepsilon \notin \{y, \mathcal{S}(y), \dots, \mathcal{S}^{p-2}(y)\}$. By transitivity, the equations entail $\mathcal{S}(ya) = \mathcal{S}(\mathcal{S}^{p-1}(y)\cdot a)$. Two cases are possible: either $\mathcal{S}^{p-1}(y) = \varepsilon$, yielding $\mathcal{S}(ya) = \mathcal{S}(a) := \varepsilon$, or the unfolding keeps going on until we find the smallest $q \geqslant p$ such that $\mathcal{S}(\mathcal{S}^{q-1}(y)\cdot a) = \mathcal{S}(\mathcal{S}^q(y)\cdot a)$ with $\mathcal{S}^q(y)\cdot a \preccurlyeq \mathcal{S}^{q-1}(y)$. Because a side is a proper prefix, that is, $\mathcal{S}(y) \prec y$, we have $\mathcal{S}^2(y) = \mathcal{S}(\mathcal{S}(y)) \prec \mathcal{S}(y)$, which, repeated, yields

$$\mathcal{S}^{q-1}(y) \preccurlyeq \cdots \preccurlyeq \mathcal{S}^{p-1}(y) \prec \cdots \prec \mathcal{S}(y) \prec y.$$

Therefore $\mathcal{S}^q(y)\cdot a \preccurlyeq y$, since $q > 0$, and $\mathcal{S}(ya) = \mathcal{S}^q(y)\cdot a$. This reasoning establishes that

$$\mathcal{S}(ya) = \begin{cases}\mathcal{S}^q(y)\cdot a, & \text{if } \exists q > 0 \text{ such that } \mathcal{S}^q(y)\cdot a \preccurlyeq y,\\ \varepsilon & \text{otherwise;}\end{cases} \tag{15.64}$$

with the additional constraint that q must be as small as possible. This is the formal derivation whose intuition was already present in the example of FIGURE 104 on the next page, where $q = 3$. This form of the definition of $\mathcal{S}$ is simpler because it does not contain an embedded call like $\mathcal{S}(\mathcal{S}(y)\cdot a)$. We can now take the lengths of each sides of the equations, leading to

$$|\mathcal{S}(ya)| = \begin{cases}|\mathcal{S}^q(y)\cdot a| = 1 + |\mathcal{S}^q(y)|, & \text{if } \exists q \text{ such that } \mathcal{S}^q(y)\cdot a \preccurlyeq y,\\ |\varepsilon| = 0 & \text{otherwise;}\end{cases}$$

If $ya \preccurlyeq x$, then$|\mathcal{S}(ya)| = \mu_x(|ya|) = \mu_x(|y|+1)$. Let $|y| := i > 0$. Then

$$\mu_x(i+1) = \begin{cases}1 + |\mathcal{S}^q(y)|, & \text{if } \exists q > 0 \text{ such that } \mathcal{S}^q(y)\cdot a \preccurlyeq y,\\ 0 & \text{otherwise;}\end{cases}$$

We need to work on $|\mathcal{S}^q(y)|$ now. From the definition of μ (15.62) on the following page, we deduce

$$\mu_x^q(|y|) = |\mathcal{S}^q(y)|, \text{ with } y \preccurlyeq x, \tag{15.65}$$

which we can prove by complete induction on q. Let us call this property $\mathcal{P}(q)$. Trivially, we have $\mathcal{P}(0)$. Let us suppose $\mathcal{P}(r)$ for all $r \leqslant q$: this is the *induction hypothesis*. Let us prove now $\mathcal{P}(r+1)$:

$$\mu_x^{r+1}(|y|) = \mu_x^r(\mu_x(|y|)) := \mu_x^r(|\mathcal{S}(y)|) \doteq |\mathcal{S}^r(\mathcal{S}(y))| = |\mathcal{S}^{r+1}(y)|,$$

where ($\doteq$) is a valid application of the induction hypothesis because $\mathcal{S}(y) \prec y$. We proved that $\mathcal{P}(r+1)$ and the induction principle entails that $\mathcal{P}(q)$ holds for all $q \geqslant 0$. Therefore, equation (15.65) allows us to refine our definition as follows, with $i > 0$:

$$\mu_x(i+1) = \begin{cases} 1 + \mu_x^q(i), & \text{if } \exists q > 0 \text{ such that } \mathcal{S}^q(y) \cdot a \preccurlyeq y, \\ 0 & \text{otherwise.} \end{cases}$$

There is one last equation in the definition of μ (15.61) on page 351 which we did not use yet: $\mathcal{S}(a) := \varepsilon$. It implies that $\mu_x(1) = \mu_x(|a|) := |\mathcal{S}(a)| := |\varepsilon| = 0$. If we try to see what happens to the previous equation defining μ when $i = 0$, we find that $\mu_x(1) = 1 + \mu_x^q(0)$. This equality is satisfied if $q = 1$ because of equation (15.63) on the current page, that is, $\mu_x(0) := -1$. Therefore, we do not need to carry separately the case $\mu_x(1)$. Now remains to find a numerical interpretation of the condition of existence of q in terms of indexes. Property "$\mathcal{S}^q(y) \cdot a \preccurlyeq y$ and $ya \preccurlyeq x$ and $|y| = i$" imply any of the following equalities:

$$y[|\mathcal{S}^q(y)|] = a \Leftrightarrow y[\mu_x^q(i)] = a \Leftrightarrow x[\mu_x^q(i)] = x[|y|] \Leftrightarrow x[\mu_x^q(i)] = x[i].$$

We can now gather everything we know about μ, with $i \geqslant 0$:

$$\mu_x(0) = -1;$$
$$\mu_x(i+1) = \begin{cases} 1 + \mu_x^q(i), & \text{if } \exists q > 0 \text{ such that } x[\mu_x^q(i)] = x[i], \\ 0 & \text{otherwise.} \end{cases}$$

where q is the smallest nonzero integer satisfying the condition. This can be further simplified into

$$\mu_x(0) = -1, \quad \mu_x(i+1) = 1 + \mu_x^q(i), \tag{15.66}$$

where $i \geqslant 0$ and $q > 0$ is the smallest integer such that one of the two following conditions hold:

- $1 + \mu_x^q(i) = 0$;
- $1 + \mu_x^q(i) \neq 0$ and $x[\mu_x^q(i)] = x[i]$.

Before implementing μ, let us start modifying the naive implementation on page 342 so it dovetails the improvement by Morris and Pratt. First, we need to retrieve some useful information about the partial successes of `prefix/2`, namely, i and $t[j..]$ as seen in FIGURE 101 on page 349. All we need is (1) to add a counter to `prefix/2`, initialised to 0, so it allows the caller to know where the mismatch, if any, occurred; (2) to return the current counter and text in case of mismatch (changes in bold):

```
find(_,     [],_) -> absent;
find(X,T=[_|U],J) ->
  case prefix(X,T,0) of
    yes      -> {factor,J};
    {no,V,I} -> find(X,U,J+1)                     % V is t[j..]
  end.

prefix(   [],    _,_) -> yes;
prefix([A|Y],[A|T],I) -> prefix(Y,T,I+1);
prefix(    _,    T,I) -> {no,T,I}.
```

The next step is make some use of the newly introduced `V` and `I`. The key idea of Morris and Pratt is to avoid if possible to retry matching `X` as a whole (see framed code above) and instead use $x[\mu_x(i)..]$, which means that we need to implement, besides μ, a function returning the suffix of a word. In the following, the call `suffix(`x`,`i`)` evaluates in $x[i..]$, assuming that $i \geqslant 0$:

```
suffix(    X,0) -> X;
suffix([_|X],I) -> suffix(X,I-1).
```

Let us not be hasty and replace the framed call above `find(X,U,J+1)` by `find(suffix(X,fail(X,I)),V,J+1)`, because the call `fail(X,I)` implements $\mu_x(i)$ and might evaluates in `-1`. Therefore, we need a `case` construct:

```
find(_,     [],_) -> absent;
find(X,T=[_|U],J) ->
  case prefix(X,T,0) of
    yes      -> {factor,J};
    {no,V,I} -> case fail(X,I) of           % fail(X,I) is μx(i)
                 -1 -> find(X,U,J+1);
                  B -> find(suffix(X,B),V,J+1)
                end
  end.
```

We can simplify this a little bit because $i = 0 \Rightarrow \mu_x(i) = -1$, hence:

```
find(_,     [],_) -> absent;
find(X,T=[_|U],J) ->
  case prefix(X,T,0) of
    yes      -> {factor,J};
    {no,_,0} -> find(X,U,J+1);
    {no,V,I} -> find(suffix(X,fail(X,I)),V,J+I)
  end.
```

We clearly see now, in bold typeface, the improvement proposed by Morris and Pratt. Unfortunately, this definition is wrong because, when adding a new recursive call, we should have checked that any invariant on the parameters is preserved. The problem here is the first argument, X, which was invariant before the addition of the improvement. Now, it changes sometimes into suffix(X,fail(X,I)) and this breaks the correct behaviour of the other recursive call, corresponding to the case when the mismatch happens at the first letter and the *original* word (not X anymore) must be slided of one index. Therefore, we must thread a copy of Word in all the calls to find/3, which becomes find/4:

```
find(Word,Text) -> find(Word,Text,0,Word).       % Copy of Word

find(_,     [],_,_) -> absent;
find(X,T=[_|U],J,W) ->                     % W is the original Word
  case prefix(X,T,0) of
    yes      -> {factor,J};
    {no,_,0} -> find(W,U,J+1,W);                % Sliding W by one
    {no,V,I} -> find(suffix(X,fail(X,I)),V,J+I,W)
  end.
```

Now remains to write a definition for fail/2. We need another function to compute $\mu_x^q(i)$, which we shall call fp/3, because it calculates a "fixed point" or, in short, a *fixpoint*. In mathematics, a fixpoint X of a function F, if it exists, satisfies the equation $F(X) = X$. Generally speaking, the fixpoint X is found by starting with an appropriate value X_0 and composing $F(X_0)$, $F^2(X_0)$ etc. until $F^n(X_0) = F^{n-1}(X_0)$, which means that $X = F^{n-1}(X_0)$. In imperative programming languages, the constructs implementing fixpoint calculations are loops whose exit conditions may be arbitrary, like while in C or repeat in Pascal, but not the for iteration. Resuming our argument, the calls to fp/3 have the shape fp(x,$x[i-1]$,$\mu_x(b)$), where the first value of b is $i-1$:

```
fail(_,0) -> -1;
fail(X,I) -> 1 + fp(X,$x[i-1]$,fail(X,I-1)).
```

The purpose of fp/3 is to accumulate by iteration calls to μ_x in the last argument, starting from $i-1$, until one of the two conditions of the definition of μ (equations (15.66) on the following page) are satisfied. We know that $q > 0$, that is why we start with the argument fail(X,I-1) (at least one call is necessary). We also need an Erlang function to compute $x[i-1]$, that is, a function accessing the nth item in a list x:

```
nth([A|_],0) -> A;
nth([_|X],N) -> nth(X,N-1).
```

So we have now

```
fail(_,0) -> -1;                                    % $\mu_x(0) = -1$
fail(X,I) -> 1 + fp(X,nth(X,I-1),fail(X,I-1)).     % $1 + \mu^q_x(i-1)$
```

The first two arguments of `fp/3` are invariant:

```
fp(_,_,  -1) -> -1;                             % $1 + \mu^q_x(i) = 0$
fp(X,B,Fail) -> case nth(X,Fail) of             % $1 + \mu^q_x(i) \neq 0$
                  B -> Fail;                    % $x[\mu^q_x(i)] = x[i]$
                  _ -> fp(X,B,fail(X,Fail))     % Try $\mu_x(\mu^q_x(i))$
                end.
```

We see here an aspect of the semantic of the `case` construct which is specific to Erlang: variable `B` appears both as a parameter of `fp/3` in `fp(X,B,Fail)` and as a case `B`. In Erlang, this means that the latter must evaluate in the same value as the former. This should be contrasted with the semantics of the `fun` constructs, which allows *shadowing*, for example, `fun(X) -> fun(X,Y) -> X end end` is valid and the body `X` refers to the `X` in `fun(X,Y)`, not in `fun(X)`. There is no shadowing with the `case` construct. In other functional programming languages, this behaviour is likely to be different, for instance, in OCaml, each variable occuring in a case (the construct is actually called `match`) shadows any other identical variable in the scope.

In summary, the Erlang module implementing the algorithm of Morris and Pratt for factor matching is

```
-module(mp).
-export([find/2]).

find(Word,Text) -> find(Word,Text,0,Word).

find(_,     [],_,_) -> absent;
find(X,T=[_|U],J,W) ->
  case prefix(X,T,0) of
    yes      -> {factor,J};
    {no,_,0} -> find(W,U,J+1,W);
    {no,V,I} -> F=fail(X,I), find(suffix(X,F),V,J+I-F,W)
  end.

prefix(   [],    _,_) -> yes;
prefix([A|X],[A|T],I) -> prefix(X,T,I+1);
prefix(    _,    T,I) -> {no,T,I}.
```

```
suffix(    X,0) -> X;
suffix([_|X],I) -> suffix(X,I-1).

% Maximum side lengths (failure function)
%
fail(_,0) -> -1;
fail(X,I) -> 1 + fp(X,nth(X,I-1),fail(X,I-1)).

nth([A|_],0) -> A;
nth([_|X],N) -> nth(X,N-1).

fp(_,_,  -1) -> -1;
fp(X,B,Fail) -> case nth(X,Fail) of
                  B -> Fail;
                  _ -> fp(X,B,fail(X,Fail))
                end.
```

Improvement. A look back at the definition of μ makes us realise that it does not depend on the text, so it is worth precomputing the values of $\mu_x(i)$ for $0 \leqslant i \leqslant n$, that is, for all prefixes of x. Moreover, we do not want to lose time in accessing the value of $\mu_x(i)$, so it would be efficient to pair it with the letter $x[i]$. For example, the table in FIGURE 105 on the preceding page would be implemented as the list

```
[{a,-1},{b,0},{a,0},{c,1},{a,0},{b,1},{a,2},{c,3}].
```

Then the programs uses this list instead of the original word. Notice that we do not need to record the value of $\mu_x(|x|)$, as this would be only needed if there was a failure *after* x, which makes no sense. (In fact, this case would precisely mean the opposite: `prefix/3` succeeded, that is, x is a factor.) Let us call `mk_fail/1` a function such that `mk_fail(`x`)`, where $|x| = m$, evaluates in the list

`[{`$x[0]$`,`$\mu_x(0)$`}, {`$x[1]$`,`$\mu_x(1)$`}, …, {`$x[m-1]$`,`$\mu_x(m-1)$`}].`

This operation is a map (see page 263):

```
mk_fail(X)         -> mk_fail(X,X,0).       % X copied for μx(i)
mk_fail(   [],_,_) -> [];
mk_fail([A|Y],X,I) -> [{A,fail(X,I)}|mk_fail(Y,X,I+1)].
```

This preprocessing on the input word is implemented now by a single call, at the beginning, whose value is shared with the copy:

```
find(Word,Text) -> W=mk_fail(Word), find(W,Text,0,W).
```

Now we need to modify `prefix/3` so that, in case of mismatch, it also returns the failure index $\mu_x(i)$:

```
prefix(        [],    _,_) -> yes;
prefix([{A,_}|X],[A|T],I) -> prefix(X,T,I+1);
prefix([{_,F}|_],    T,I) -> {no,T,I,F}.           % F is μx(i)
```

Now `find/3` becomes (differences in bold typeface):

```
find(_,     [],_,_) -> absent;
find(X,T=[_|U],J,W) ->
  case prefix(X,T,0) of
    yes        -> {factor,J};
    {no,_,0,_} -> find(W,U,J+1,W);                          % Here
    {no,V,I,F} -> find(suffix(X,F),V,J+I,W)             % and here
  end.
```

A closer look back at `mk_fail/3` and `fail/2` gives us a clue on how to improve further the efficiency. On the one hand, we see that the call `fail(X,I)` relies on the value of the recursive call `fail(X,I-1)`. On the other hand, we observe that `mk_fail/3` computes `fail(X,I)` for increasing values of `I`. The loss of efficiency comes thus from `mk_fail/3` not reusing the prior value of `fail(X,I)`, that is, `fail(X,I-1)`, and not passing it along to `fp/3` as the third argument, that is, the original value to compute the fixpoint. Here is the modified version of `mk_fail/1` and `mk_fail/3`, the latter becoming `mk_fail/4`:

```
mk_fail(X)              -> mk_fail(X,X,0,none).
mk_fail(   [],_,_,   _) -> [];
mk_fail([A|Y],X,I,Prev) ->
  Fail = case I of
           0 -> -1;
           _ -> 1 + fp(X,nth(X,I-1),Prev)
         end,
  [{A,Fail}|mk_fail(Y,X,I+1,Fail)].
```

Notice how we had to initialise the new last argument of `mk_fail/4` with a dummy value `none` because there is no prior value of a call to `fail/2` at this point. The value of the argument `I` is `0` if, and only if, `Prev` is `none`, thus we could have chosen any value other than `none` since we discriminate on the index `I` only. The `case` construct shows how we duplicated one rewrite step of a call to `fail/2`, in order to sneak `Prev` instead of `fail(X,I-1)` to `fp/3`. This was the whole point of this transformation.

Another look to the brand new `mk_fail/4` reveals that `nth(X,I-1)` could be avoided because it evaluates in the letter in `X` just before `A`. Just like we kept the value of the call to `fail/2` on the previous index, `Prev`, we could keep the corresponding letter as well. In other words, instead of

```
[{A,Fail}|mk_fail(Y,X,I+1,Fail)].
```

we would write something equivalent to

```
[{A,Fail}|mk_fail(Y,X,I+1,{A,Fail})].
```

Here is how:

```
mk_fail(   [],_,_,   _) -> [];
mk_fail([A|Y],X,I,Prev) ->
  Cur = case Prev of
          none     -> {A,-1};
          {B,Fail} -> {A,1 + fp(X,B,Fail)}
        end,
  [Cur|mk_fail(Y,X,I+1,Cur)].
```

Notice how we discriminate now on `Prev` in the `case` construct, instead of `I`. Furthermore, since the call to `nth/2` disappeared, as intended, there is no use for `I` anymore. Just let get rid of it:

```
mk_fail(   [],_,   _) -> [];
mk_fail([A|Y],X,Prev) ->
  Cur = case Prev of
          none     -> {A,-1};
          {B,Fail} -> {A,1 + fp(X,B,Fail)}
        end,
  [Cur|mk_fail(Y,X,Cur)].
```

After several code transformations, it is important to look back and make sure that they are correct and, if so, whether better alternatives are possible. In our case, a fresh look at the result of the last transformation may bring to the fore something quite obvious, actually: that the `case` construct is evaluated for each letter in the word, whilst the case of `Prev` evaluating to `none` happens only once, at the first call when the word is not empty. In other words, the test conveyed by the `case` is useless $m - 1$ times. Even if in our time model we do not take into account the delay of a `case`, but only function calls, it surely matters in practice. The way to avoid this situation is simple: let us move the burden of handling the special situation leading to `none` to `mk_fail/1`. Since the latter actually introduces `none`, let us simply avoid doing so:

```
mk_fail(     []) -> [];
mk_fail(X=[A|Y]) -> Fst={A,-1}, [Fst|mk_fail(Y,X,Fst)].

mk_fail(   [],_,       _) -> [];
mk_fail([A|Y],X,{B,Fail}) -> Cur = {A,1 + fp(X,B,Fail)},
                             [Cur|mk_fail(Y,X,Cur)].
```

Now `mk_fail/1` handles the first letter and `mk_fail/3` processes all the remaining letters. The final improved version is then

```
-module(mp_opt).
-export([find/2]).

find(Word,Text) -> W=mk_fail(Word), find(W,Text,0,W).

find(_,     [],_,_) -> absent;
find(X,T=[_|U],J,W) ->
  case prefix(X,T,0) of
    yes        -> {factor,J};
    {no,_,0,_} -> find(W,U,J+1,W);
    {no,V,I,F} -> find(suffix(X,F),V,J+I-F,W)
  end.

prefix(       [],    _,_) -> yes;
prefix([{A,_}|X],[A|T],I) -> prefix(X,T,I+1);
prefix([{_,F}|_],    T,I) -> {no,T,I,F}.

suffix(    X,0) -> X;
suffix([_|X],I) -> suffix(X,I-1).

% Preprocessing maximum side lengths (failures)
%
fail(_,0) -> -1;
fail(X,I) -> 1 + fp(X,nth(X,I-1),fail(X,I-1)).

nth([A|_],0) -> A;
nth([_|X],N) -> nth(X,N-1).

fp(_,_,  -1) -> -1;
fp(X,B,Fail) -> case nth(X,Fail) of
                  B -> Fail;
                  _ -> fp(X,B,fail(X,Fail))
                end.

mk_fail(     []) -> [];
mk_fail(X=[A|Y]) -> Fst={A,-1}, [Fst|mk_fail(Y,X,Fst)].
mk_fail(   [],_,        _) -> [];
mk_fail([A|Y],X,{B,Fail}) -> Cur = {A,1 + fp(X,B,Fail)},
                             [Cur|mk_fail(Y,X,Cur)].
```

Best Delay. Of course, the last improvement results in the best case being slower because this algorithm now features two stages, the preprocessing of the word (`mk_fail/1`) and the proper matching (`find/4`), and we just made the preprocessing a prerequisite in all circumstances. First, let us note $\mathcal{D}_m^{\texttt{mk\_fail/1}}$ the delay of the calls `mk_fail(`x`)`, where the word x has length m. If we only count the calls to `mk_fail/1` and `mk_-fail/3`, then $\mathcal{D}_0^{\texttt{mk\_fail/1}} = 1$ and $m > 0$ imply $\mathcal{D}_m^{\texttt{mk\_fail/1}} > m+1$, because the word is always processed in its entirety. Now remain to be accounted for the calls to `fp/3`, which number at least $m - 1$, that is, one for each letter in the word. Because we are in pursuit of the best case and `fp/3` is recursive, we should think first about a class of inputs which lead to minimise or avoid altogether recursive calls. A look back at the definition of `fp/3` reveals that this is possible in two cases: either the third argument `Fail` is `-1` or `nth(X,Fail)` evaluates in the second argument `B`. For instance, the first call to `fp/3` has the shape `fp(`x`,`$x[0]$`,-1)`, so it results in one rewrite in the value `-1`. In other words, this situation corresponds to $i = 0$ and $q = 1$ and $1 + \mu_x^q(i) = 0$ in equation (15.66) on this page. The following calls to `fp/3` are the quickest when the first call to `nth/2` results in `B` *with the shortest delay*. First, what does it mean that the first call results in `B`? This situation corresponds to the condition $q = 1$ and $1+\mu_x^q(i) \neq 0$ and $x[\mu_x^q(i)] = x[i]$ in equation (15.66) on the current page, if $x[i]$ corresponds to `B` and $\mu_x^q(i)$ to `Fail`. In other words, when examining the letters of x from left to right, we find that each letter extends the maximum side of the previous prefix. This means that x is made of the repetition of the same letter, for example, $x = \texttt{a}^m$, which results in

$$\texttt{mk\_fail(}x\texttt{)} \twoheadrightarrow \texttt{[\{a,-1\},\{a,0\},\{a,1\},...,\{a,}m-1\texttt{\}]}.$$

But we forgot to check the constraint that the calls `nth(X,Fail)` have the minimum delay individually or in sum. The delay of `nth(`x`,`j`)` is $j+1$ and the calls are performed on the previous letter's failure index, thus j, that is, `Fail`, varies from 0 (failure index of $x[1]$) to $m - 3$ (failure index of $x[m - 2]$). Therefore the problem with the previous example is that the additional delay of all the calls to `nth/2` is $\sum_{j=0}^{m-3}(j + 1) = (m - 2)(m - 1)/2$. This quadratic delay is enough ground to look for a better solution. The best total delay for the calls to `nth/2` is achieved when all the calls have their best delay, which is 1. This means that `Fail` is always `0` and the total additional delay is $m - 2$. This happens when the first letter of the word differs from all the following, for example, $x =$ `abbcbd` and

`mk_fail(`x`)` $\twoheadrightarrow$ `[{a,-1},{b,0},{b,0},{c,0},{b,0},{d,0}].`

Now that the minimality condition on `nth/2` is satisfied, what becomes of the delay of the rest of `fp/3`? We cannot have `nth(X,Fail)` to evaluate in `B` because this would entail in `mk_fail/3` the next value of `Fail` to be $1 + 0 = 1$, breaking our minimality condition on the next call to `fp/3`. Therefore, a recursive call to `fp/3` is unavoidable: `fp(X,B,fail(X,Fail))`, where `Fail` is actually `0`. The value of the call `fail(X,0)` is `-1`, with a delay of 1, so we have to consider `fp(X,B,-1)` in turn. This is also easy: this is `-1` in one rewrite too. Therefore the recursive call has the delay 2, so the total delay of `fp/3` is $2(m-2)+1 = 2m-3$, the 1 coming from the first call to `fp/3` always being equivalent to `fp(X,B,-1)`, entailing no call to `nth/2`. This is a linear delay, thus a definite improvement over our prior attempt. As a summary, the best case happens when the first letter of the word of length m is unique and the delay for the preprocessing is then

$$\mathcal{B}_m^{\texttt{mk\_fail/1}} = (m+1) + (2m-3) = 3m - 2.$$

We can now shift our focus on the second stage of the algorithm, that is, the search itself by means of the call `find(W,Text,0,W)`. To reach the best delay, we can, as usual, first look to minimise the number of recursive calls. While examining the definition of `find/4`, we must take care not to conclude that the best case is when the text is empty. Indeed, let us remember that we must assume that the size of the input, here the text, is given and we have no way to know what it is, in particular, whether it is 0 or not. The other interpretation of ending with the atom `absent` is, obviously, when all the text has been searched for an occurrence of the word, which is not what we are seeking for in the best case. The other way to minimise the number of recursive calls is to have the first call `prefix(X,T,0)` evaluate in the atom `yes`, that is, the word is a prefix of the text. So let us turn our attention to `prefix/3` and determine its delay when its final value is `yes`. There must be exactly one recursive call for each letter in the word before `yes` is final. If we note $\mathcal{B}_{m,n}^{\texttt{prefix}}$ the best delay of the call `prefix(`x`,`t`)`, where $|x| = m$ and $|t| = n$, we have

$$\mathcal{B}_{m,n}^{\texttt{prefix}} = m + 1.$$

As a conclusion, if we note $\mathcal{B}_{m,n}^{\texttt{find}}$ the best delay of the the call `find(`x`,`t`)`,

$$\mathcal{B}_{m,n}^{\texttt{find}} = 1 + \mathcal{B}_m^{\texttt{mk\_fail/1}} + \mathcal{B}_{m,n}^{\texttt{find/4}} = 1 + (3m-2) + (1 + \mathcal{B}_{m,n}^{\texttt{prefix}}) = 4m + 1.$$

This happens when the first letter of the word differs from all the following and the word is a prefix of the test. We can see that, without the preprocessing of the word, the best delay of the Morris-Pratt would be $m+3$, which is precisely the best delay of the naive algorithm (see page 344).

What if we want to compare both algorithms in more abstract terms by comparing the number of comparisons they perform, in the best case or otherwise? This question is equivalent to compare the number of times the second clause of `prefix/2` is used against the number of times the second clause of `prefix/3` is used plus the number of times the `case` construct is executed during the preprocessing, because it relies on letter comparisons as well. If we ignore the latter, in the best case, these numbers must be equal, as the best cases for both algorithms agree on the word being a prefix of the text; more precisely, m comparisons are needed. As we already know, in the best case, `nth/2` is called $m-2$ times in the `case`, so the total number of comparisons in the best case is $m+(m-2) = 2m-2$ for the Morris-Pratt algorithm, versus m for the naive algorithm.

Exercise. Prove that the Morris-Pratt algorithm makes at most $2(n+m-2)$ comparisons to find a word or fail.

Knuth-Morris-Pratt Algorithm. The algorithm we present now is an improvement due to Knuth, Morris and Pratt (1977), based on avoiding situations which lead to certain letter comparison failures. Consider again figure FIGURE 101 on page 349. The key observation is that if $x[k] =$ `a` then the slide would lead to an immediate mismatch because $t[j] \neq x[k]$. In order to avoid it, we need to use the longest side of $x[0..i-1]$ *which is not followed by* `a`. It is called the *maximum tagged side* of $x[0..i-1]$ in x. Let us note $\overline{\mathcal{S}}_x(y)$ the maximum tagged side of any proper prefix y of x. If $ya \preccurlyeq x$, the letter a constrains the definition of the tagged side, as we must have $\overline{\mathcal{S}}_x(y) \cdot a \not\preccurlyeq y$. What if $y = x$, then? In this case, there is no right-context, like the letter a above, to constrain the maximum tagged side. In this case, we still can take the maximum side, that is, $\overline{\mathcal{S}}_y(y) = \mathcal{S}(y)$. More precisely, if the maximum side of y is not followed by a, then it is also the maximum tagged side we are looking for. In other words:

$$\overline{\mathcal{S}}_x(y) = \mathcal{S}(y), \text{ if } ya \preccurlyeq x \text{ and } \mathcal{S}(y) \cdot a \not\preccurlyeq y.$$

If the maximum side of y is followed by a we must take the maximum tagged side of the maximum side:

$$\overline{\mathcal{S}}_x(y) = \overline{\mathcal{S}}_x(\mathcal{S}(y)), \text{ if } ya \preccurlyeq x \text{ and } \mathcal{S}(y) \cdot a \preccurlyeq y.$$

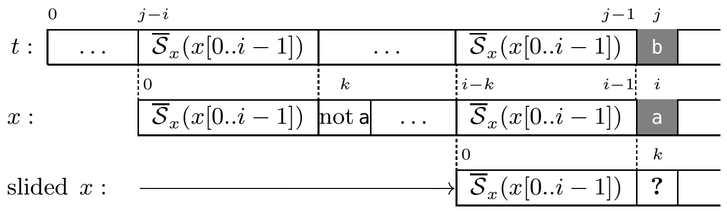

0
$j-i$
$j-1$
j
t :
...
$\overline{\mathcal{S}}_x(x[0..i-1])$
...
$\overline{\mathcal{S}}_x(x[0..i-1])$
b
0
k
$i-k$
$i-1$
i
x :
$\overline{\mathcal{S}}_x(x[0..i-1])$
not a
...
$\overline{\mathcal{S}}_x(x[0..i-1])$
a
0
k
slided x :
$\overline{\mathcal{S}}_x(x[0..i-1])$
?

FIGURE 106: Rationale for Knuth-Morris-Pratt algorithm

By extension, if $x = y$, we take the maximum side:

$$\overline{\mathcal{S}}_y(y) = \mathcal{S}(y).$$

In summary, assuming $ya \preccurlyeq x$ and $y \neq \varepsilon$

$$\overline{\mathcal{S}}_y(y) := \mathcal{S}(y);$$

$$\overline{\mathcal{S}}_x(y) := \begin{cases} \overline{\mathcal{S}}_x(\mathcal{S}(y)), & \text{if } \mathcal{S}(y) \cdot a \preccurlyeq y, \\ \mathcal{S}(y), & \text{otherwise.} \end{cases}$$

FIGURE 106 on page 365 summarises the improvement of Knuth, Morris and Pratt. Consider the example in FIGURE 102 on page 350 and compare it with FIGURE 107 on page 365, where the same word is searched in the same text taking into account the improvement on the sides: 14 comparisons are performed, instead of 17. The table comparing the

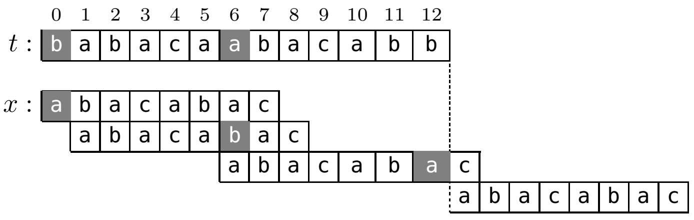


FIGURE 107: Knuth-Morris-Pratt algorithm at work (no match found)

values of the two failure functions for the same word abacabac can be found in FIGURE 108 on page 365. By taking the length of each side of the equations,

$$|\overline{\mathcal{S}}_x(y)| = \begin{cases} |\mathcal{S}(y)| & \text{if } x = y \text{ or } \mathcal{S}(y) \cdot a \not\preccurlyeq y \\ |\overline{\mathcal{S}}_x(\mathcal{S}(y))| & \text{otherwise} \end{cases}$$

Let $\kappa_x(i)$ be the length of the tagged maximum side of $x[1 \dots i]$:

$$\kappa_x(i) = |\overline{\mathcal{S}}_x(x[1...i])| \qquad 1 \leqslant i \leqslant |x|$$

or, equivalently,

$$\kappa_x(|y|) = |\overline{\mathcal{S}}_x(y)| \qquad y \preccurlyeq x$$

Therefore

$$\kappa_x(|y|) = \begin{cases} \mu_x(|y|) & \text{if } x = y \text{ or } \mathcal{S}(y) \cdot a \not\preccurlyeq y \\ \kappa_x(|\mathcal{S}(y)|) & \text{otherwise} \end{cases}$$

that is to say

$$\kappa_x(|y|) = \begin{cases} \mu_x(|y|) & \text{if } x = y \text{ or } \mathcal{S}(y) \cdot a \not\preccurlyeq y \\ \kappa_x(\mu_x(|y|)) & \text{otherwise} \end{cases}$$

If $|y| = i$, we can write instead

$$\kappa_x(i) = \begin{cases} \mu_x(i) & \text{if } x = y \text{ or } \mathcal{S}(y) \cdot a \not\preccurlyeq y \\ \kappa_x(\mu_x(i)) & \text{otherwise} \end{cases}$$

The condition can also be rewritten in terms of μ_x and i, just as we did with the Morris-Pratt algorithm, in FIGURE 349. Let $|y| = i$, then

$$\mathcal{S}(y) \cdot a \not\preccurlyeq y \Leftrightarrow x[\mu_x(i) + 1] \neq x[i + 1]$$
$$x = y \Leftrightarrow |x| = i$$

So, finally, for $1 \leqslant i \leqslant |x|$,

$$\kappa_x(i) = \begin{cases} \mu_x(i) & \text{if } i = |x| \text{ or } x[\mu_x(i) + 1] \neq x[i + 1] \\ \kappa_x(\mu_x(i)) & \text{otherwise} \end{cases}$$

which allows us to naturally extends κ_x on 0: $\kappa_x(0) = \mu_x(0) = -1$.

Before going further, let us check by hand the following values of $\kappa_x(i)$ and compare them to $\mu_x(i)$: we must always have $\kappa_x(i) \leqslant \mu_x(i)$, since we plan an optimisation.

x		a	b	c	a	b	a	b	c	a	c
i	0	1	2	3	4	5	6	7	8	9	10
$\mu_x(i)$	−1	0	0	0	1	2	1	2	3	4	0
$\kappa_x(i)$	−1	0	0	−1	0	2	0	0	−1	4	0

One difference between μ_x and κ_x is that, in the worst case, there is always an empty maximum side ε, that is, $\mu_x(i) = 0$, whereas there may

x	a	b	a	c	a	b	a	c
i	0	1	2	3	4	5	6	7
$\mu_x(i)$	−1	0	0	1	0	1	2	3
$\kappa_x(i)$	−1	0	−1	1	−1	0	−1	1

FIGURE 108: Failure functions of abacabac compared

be no maximum tagged side at all, that is, $\kappa_x(i) = -1$. For instance, the prefix `abc` of x has an empty maximum side, that is, $\mu_x(3) = 0$, but has no maximum tagged side, that is, $\kappa_x(3) = -1$, since $x[1] = x[4]$.

This definition of κ_x relies on μ_x, more precisely, the computation of $\kappa_x(i)$ requires $\mu_x^p(i)$, that is, values $\mu_x(j)$ with $j < i$, since

$$\mu_x(0) = -1 \qquad \mu_x(i) = 1 + \mu_x^k(i-1) \qquad 1 \leqslant i$$

where k is the smallest non-zero integer such that

- either $1 + \mu_x^k(i-1) = 0$ or $x[1 + \mu_x^k(i-1)] = x[i]$

or, equivalently,

$$\mathcal{S}(ya) = \begin{cases} \mathcal{S}^k(y) \cdot a & \text{if } \mathcal{S}^k(y) \cdot a \preccurlyeq y \\ \varepsilon & \text{if } \mathcal{S}^{k-1}(y) = \varepsilon \end{cases}$$

where k is the smallest non-zero integer such that $\mathcal{S}^k(y) \cdot a \preccurlyeq y$ or $\mathcal{S}^{k-1}(y) = \varepsilon$.

Therefore, let us find another definition of $\mathcal{S}(ya)$ which relies on $\mathcal{S}(y)$ but *not* on $\mathcal{S}^q(y)$ with $2 \leqslant q$. This can be achieved by considering again the figure

$$y \cdot a : \boxed{\mathcal{S}(y)}\boxed{b}\boxed{\qquad\qquad\qquad}\boxed{\mathcal{S}(y)}\boxed{a}$$

Indeed, if $a = b$ then $\mathcal{S}(ya) = \mathcal{S}(y)$. Else, $a \neq b$, and thus the maximum side can be found among the tagged sides of $\mathcal{S}(y)$, otherwise $\mathcal{S}(ya) = \varepsilon$.

$$\mathcal{S}(ya) = \begin{cases} \overline{\mathcal{S}}_x^q(\mathcal{S}(y)) \cdot a & \text{if } \overline{\mathcal{S}}_x^q(\mathcal{S}(y)) \cdot a \preccurlyeq y \\ \varepsilon & \text{if } y = \varepsilon \text{ or } \overline{\mathcal{S}}_x^{q-1}(\mathcal{S}(y)) = \varepsilon \end{cases}$$

where $ya \preccurlyeq x$ and q is the smallest integer such that $\overline{\mathcal{S}}_x^q(\mathcal{S}(y)) \cdot a \preccurlyeq y$ or $\overline{\mathcal{S}}_x^{q-1}(\mathcal{S}(y)) = \varepsilon$.

By taking the lengths and letting $0 \leqslant i \leqslant |x| - 1$ and $|y| = i$,

$$\mu_x(0) = -1$$
$$\mu_x(i+1) = \begin{cases} \kappa_x^q(\mu_x(i)) + 1 & \text{if } x[\kappa_x^q(\mu_x(i)) + 1] = x[i+1] \\ 0 & \text{if } i = 0 \text{ or } \kappa_x^{q-1}(\mu_x(i)) = 0 \end{cases}$$

where q is the smallest integer such that

- either $x[\kappa_x^q(\mu_x(i)) + 1] = x[i+1]$
- or $\kappa_x^{q-1}(\mu_x(i)) = 0$

Since $\kappa_x(i) = -1$ if and only if $i = 0$, we have

$$\begin{aligned} \kappa_x^{q-1}(\mu_x(i)) = 0 &\Leftrightarrow \kappa_x(\kappa_x^{q-1}(\mu_x(i))) = \kappa_x(0) \Leftrightarrow \kappa_x^q(\mu_x(i)) = -1 \\ &\Leftrightarrow \kappa_x^q(\mu_x(i)) + 1 = 0 \end{aligned}$$

Therefore we can simplify the new definition of μ_x as

$$\begin{aligned} \mu_x(0) &= -1 \\ \mu_x(1) &= 0 \\ \mu_x(i+1) &= \kappa_x^q(\mu_x(i)) + 1 \qquad 1 \leqslant i \leqslant |x| - 1 \end{aligned}$$

where q is the smallest integer such that

- either $x[\kappa_x^q(\mu_x(i)) + 1] = x[i+1]$
- or $\kappa_x^q(\mu_x(i)) + 1 = 0$

Chapter 16

Answers to Exercises

Joining and Reversing

[See questions on page 60.]

Question 1. An auxiliary function `len__/2` is needed to hold the current length from the beginning of the list. Accordingly, it has to be initialised to `0` in the first call.

```
len_tf(L)         $\xrightarrow{\alpha}$ len__(L,0).
len__(    [],N)   $\xrightarrow{\beta}$ N;
len__([_|L]),N)   $\xrightarrow{\gamma}$ len__(L,N+1).
```

Clause α is used only once, at the start, clause β, at the very end, and clause γ is used for each item in the list, that is, n times. In total: $n+2$ rewrites.

Question 2. Here is how to find the penultimate item in a list:

```
penul([I,_]) -> I;                             % Use a comma
penul([_|L]) -> penul(L).               % Use a vertical bar
```

We have to make sure that the calls `penul([])` and `penul([`I`])`, for any I, fail, whilst all the others succeed. When faced to choose among an infinite set of test values, select those which seems extreme cases. Here, this method is not useful, though, because the function is fully polymorphic in the type of the items, that is, it does not depend on the nature of the lists elements. This definition is in tail form because each body is in tail form: the first because it is a value, the second because it is a call not embedded in another call and not containing other calls. The number of rewrites to reach a value is number of items before and including the penultimate, that is, $n-1$ if the list contains n items.

Question 3. Here a way to repeat the first item, while ignoring the others:

```
rep_fst([I|L])  -> [I|repeat(I,L)].
repeat(_,   []) -> [];
repeat(I,[_|L]) -> [I|repeat(I,L)].
```

Notice that these are *not* definitions in tail form, because the calls `repeat(I,L)` are embedded into a list in the first and last clauses. A bad design would be to first compute the length of the list and then construct a list of this length containing the first item, because it would require two complete traversals of the list instead of one, thus about 50% more time, since the number of rewrites here is $n+2$ for an input of n items.

The following version uses temporarily more memory but is shorter. The number of rewrites to find a value is also $n+1$.

```
rep_fst(     []) -> [];
rep_fst(    [I]) -> [I];
rep_fst([I,_|L]) -> [I|rep_fst([I|L])].     % Two | instead of 1
```

Question 4. The first approach that perhaps should spring up is to reuse `rep_fst/1` and the list reversing function `rev/1`:

```
rep_lst(L) -> rep_fst(rev(L)).

rev(L)            -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

The number of rewrites to compute a call is $2n+3$. Another method consists in finding the last item and repeating:

```
rep_lst([]) -> [];
rep_lst(L) -> rep_fst(last(L)).

last(  [I]) -> I;
last([_|L]) -> last(L).
```

The number of rewrites to reach the result is $2n+2$. What should be avoided here is the temptation of computing the length of the list with the intent of rebuilding one of same length, filled with the last item. This would lead to something in the following lines:

```
rep_lst([I|L])      -> len_and_make([I|L],0).

len_and_make(  [I],N) -> make(N+1,I);
len_and_make([_|L],N) -> len_and_make(L,N+1).

make(0,_)             -> [];
make(N,I)            -> [I|make(N-1,I)].
```

The number of rewrites is $1 + n + (n + 1) = 2n + 2$ but this program is really too long for solving such a simple problem, so previous approaches must be preferred instead.

Question 5. Let us label the rewrite rules with some Greek letters to distinguish them:

`rev_bis(   [])` $\xrightarrow{\alpha}$ `[];`
`rev_bis([I|L])` $\xrightarrow{\beta}$ `join(rev_bis(L),[I],[]).`

`join(   [],Q,   [])` $\xrightarrow{\gamma}$ `Q;`
`join(   [],Q,[E|A])` $\xrightarrow{\delta}$ `join([],[E|Q],A);`
`join([E|P],Q,    A)` $\xrightarrow{\epsilon}$ `join(P,Q,[E|A]).`

We deduce the trace of `rev_bis([3,2,1])` in flat representation as follows (we skip the abstract syntax trees):

`rev_bis([3,2,1])`
$\xrightarrow{\beta}$ `join(rev_bis([2,1]),[3],[])`
$\xrightarrow{\beta}$ `join(join(rev_bis([1]),[2],[]),[3],[])`
$\xrightarrow{\beta}$ `join(join(join(rev_bis([]),[1],[]),[2],[]),[3],[])`
$\xrightarrow{\alpha}$ `join(join(join([],[1],[]),[2],[]),[3],[])`
$\xrightarrow{\gamma}$ `join(join([1],[2],[]),[3],[])`
$\xrightarrow{\epsilon}$ `join(join([],[2],[1]),[3],[])`
$\xrightarrow{\delta}$ `join(join([],[1,2],[]),[3],p[)`
$\xrightarrow{\gamma}$ `join([1,2],[3],[])`
$\xrightarrow{\epsilon}$ `join([2],[3],[1])`
$\xrightarrow{\epsilon}$ `join([],[3],[2,1])`
$\xrightarrow{\delta}$ `join([],[2,3],[1])`
$\xrightarrow{\delta}$ `join([],[1,2,3],[])`
$\xrightarrow{\gamma}$ `[1,2,3].`

Delay

[See questions on page 73.]

Question 1. We are given

`rev_ter(   [])` $\xrightarrow{\alpha}$ `[];`
`rev_ter([I|L])` $\xrightarrow{\beta}$ `join_tf(rev_ter(L),[I]).`

`join_tf(P,Q)        ` $\xrightarrow{\gamma}$ `join(P,Q,[]).`
`join(   [],Q,   [])` $\xrightarrow{\delta}$ `Q;`
`join(   [],Q,[I|A])` $\xrightarrow{\epsilon}$ `join([],[I|Q],A);`
`join([I|P],Q,    A)` $\xrightarrow{\zeta}$ `join(P,Q,[I|A]).`

Let $\mathcal{D}_n^{\texttt{rev\_ter}}$ be the delay of the call `rev_ter(`L`)`, where the list L holds n items. The definitions yield the equations

$$\mathcal{D}_0^{\texttt{rev\_ter}} \overset{\alpha}{=} 1, \quad \mathcal{D}_{n+1}^{\texttt{rev\_ter}} \overset{\beta}{=} 1 + \mathcal{D}_n^{\texttt{rev\_ter}} + \mathcal{D}_n^{\texttt{join\_tf}}.$$

We proved earlier that $\mathcal{D}_n^{\texttt{join\_tf}} = 2n + 2$, therefore

$$\begin{aligned}\mathcal{D}_{n+1}^{\texttt{rev\_ter}} &= 2n + 3 + \mathcal{D}_n^{\texttt{rev\_ter}}\\ \sum_{k=0}^{n-1}\mathcal{D}_{k+1}^{\texttt{rev\_ter}} &= \sum_{k=0}^{n-1}(2k+3) + \sum_{k=0}^{n-1}\mathcal{D}_k^{\texttt{rev\_ter}}\\ \mathcal{D}_n^{\texttt{rev\_ter}} &= 2 \cdot \frac{n(n-1)}{2} + 3n + \mathcal{D}_0^{\texttt{rev\_ter}} \overset{\alpha}{=} n^2 + 2n + 1 = (n+1)^2.\end{aligned}$$

Question 2. Skipped.

Question 3. We are given

```
join_3a(P,Q,R) -> join(P,join(Q,R)).
join_3b(P,Q,R) -> join(join(P,Q),R).
```

The proved earlier on page 66 that $\mathcal{D}_{n,m}^{\texttt{join}} = n+1$, where n and m are, respectively, the length of the first and second lists. We straightforwardly deduce

$$\begin{aligned}\mathcal{D}_{p,q,r}^{\texttt{join\_3a}} &= 1 + \mathcal{D}_{q,r}^{\texttt{join}} + \mathcal{D}_{p,q+r}^{\texttt{join}} = 1 + (q+1) + (p+1) = p + q + 3;\\ \mathcal{D}_{p,q,r}^{\texttt{join\_3b}} &= 1 + \mathcal{D}_{p,q}^{\texttt{join}} + \mathcal{D}_{p+q,r}^{\texttt{join}} = 1 + (p+1) + (p+q+1) = 2p + q + 3\\ &= \mathcal{D}_{p,q,r}^{\texttt{join\_3a}} + p \geqslant \mathcal{D}_{p,q,r}^{\texttt{join\_3a}}.\end{aligned}$$

Therefore, never use the call `join(join(`P`,`Q`),`R`)`.

Filtering Out

[See questions on page 113.]

Question 1. Skipped.

Question 2. The conceptual difference between `rm_fst/2` and the sought `rm_all/2` is simply that the latter must not stop after finding the first occurrence, but instead traverse the whole list to find more occurrences—and ignore them in the reconstructed list. Hence

```
rm_all(_,   []) -> [];
rm_all(I,[I|L]) -> rm_all(I,L);
rm_all(J,[I|L]) -> [I|rm_all(J,L)].
```

Following the previous remark, there is no worst case and there is no need to rely on equations to compute the delay: $\mathcal{D}_n^{\texttt{rm\_all}} = n + 1$. In the same vein, the number of pushes if the number of found items. A version in tail form requires a list accumulator in which to save the visited items which are not the one sought after, and, therefore, it needs a final reversal:

```
rm_all_tf(I,L) -> rm_all_tf__(I,L,[]).

rm_all_tf__(_,   [],A) -> rev(A);
rm_all_tf__(I,[I|L],A) -> rm_all_tf__(I,L,A);
rm_all_tf__(I,[J|L],A) -> rm_all_tf__(I,L,[J|A]).
```

Question 3.

$$
\begin{array}{lll}
\texttt{rm\_lst3(\_,   [])} & \xrightarrow{\alpha} & \texttt{[];}\\
\texttt{rm\_lst3(I,[I|P])} & \xrightarrow{\beta} & \texttt{aux3(I,P,P);}\\
\texttt{rm\_lst3(I,[J|P])} & \xrightarrow{\gamma} & \texttt{[J|rm\_lst3(I,P)].}\\
\texttt{aux3(\_,   [],P)} & \xrightarrow{\delta} & \texttt{P;}\\
\texttt{aux3(I,[I|\_],P)} & \xrightarrow{\epsilon} & \texttt{[I|rm\_lst3(I,P)];}\\
\texttt{aux3(I,[\_|Q],P)} & \xrightarrow{\zeta} & \texttt{aux3(I,Q,P).}
\end{array}
$$

If the item is missing, the sequential search fails as usual with a delay of $n+1$. Otherwise, let us name $x_1 < x_2 < \cdots < x_p$ the indices of all the occurrences of the item in the list. Then, the trace is

$$\gamma^{x_1} \cdot \prod_{k=2}^{p}(\beta\zeta^{x_k - x_{k-1}-1})(\epsilon\gamma^{x_k - x_{k-1}-1}) \cdot (\beta\zeta^{n-x_p-1}\delta).$$

The length of the trace is therefore

$$x_1 + 2\sum_{k=2}^{p}(x_k - x_{k-1}) + n - x_p + 1 = n + x_p - x_1 + 1.$$

oIn other words, if the position of the first occurrence is noted f and the position of the last is k, we find that $\mathcal{D}_{n,f,k}^{\texttt{rm\_lst3}} = n+k-f+1$. We deduce that the best delay happens when $k - f + 1 = p$, that is, when all the occurrences are consecutive: we have $\mathcal{B}_{n,p}^{\texttt{rm\_lst3}} = n + p$. The worst case occurs when $f = 0$ and $k = n - 1$, that is, when there is at least one occurrence at the head and one at the end: $\mathcal{W}_{n}^{\texttt{rm\_lst3}} = n + (n-1) + 1 = 2n$. We can check that when the list only contains occurrences of the item, best and worst delays concur in $2n$. The average delay when the item is present:

$$\mathcal{A}_{n}^{\texttt{rm\_lst3}} := \frac{2}{n(n+1)}\sum_{f=0}^{n-1}\sum_{k=f}^{n-1}\mathcal{D}_{n,f,k}^{\texttt{rm\_lst3}} = \frac{2}{n(n+1)}\sum_{f=0}^{n-1}\sum_{k=f}^{n-1}(n - f + k + 1)$$

$$
\begin{aligned}
&= n+1+\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(-f(n-f)+\sum_{k=f}^{n-1}k\right)\\
&= n+1+\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(f^2-nf+\left(\sum_{k=0}^{n-1}k-\sum_{k=0}^{f-1}k\right)\right)\\
&= n+1+\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(f^2-nf+\left(\frac{n(n-1)}{2}-\frac{f(f-1)}{2}\right)\right)\\
&= n+1+\frac{2}{n(n+1)}\cdot\frac{n^2(n-1)}{2}\\
&\quad+\frac{2}{n(n+1)}\sum_{f=0}^{n-1}\left(\frac{1}{2}f^2-\frac{2n-1}{2}f\right)\\
&= n+1+\frac{n(n-1)}{n+1}\\
&\quad+\frac{1}{n(n+1)}\sum_{f=0}^{n-1}f^2-\frac{2}{n(n+1)}\cdot\frac{2n-1}{2}\sum_{f=0}^{n-1}f\\
&= n+1+\frac{n(n-1)}{n+1}\\
&\quad+\frac{1}{n(n+1)}\cdot\frac{(n-1)n(2n-1)}{6}-\frac{2n-1}{n(n+1)}\cdot\frac{n(n-1)}{2}\\
&= \frac{4}{3}n+\frac{2}{3}.
\end{aligned}
$$

The complete table is now found in FIGURE 109 on page 374. On a last note, clause δ allows the output to share with the input the longest

	Delay				
	Item				
Function	Present	Absent	Best	Worst	Average
`rm_lst/2`	$3n-k+4$	$3n+6$	$2n+5$	$3n+6$	$\frac{5}{2}n+\frac{9}{2}$
`rm_lst1/2`	$3n-f-k+4$	$n+1$	$n+1$	$3n+4$	$2n+5$
`rm_lst_tf/2`	$2n+4$	$2n+3$	$2n+3$	$2n+4$	$2n+4$
`rm_lst2/2`	$2n-k+1$	$n+1$	$n+1$	$2n+1$	$\frac{3}{2}n+\frac{3}{2}$
`rm_lst3/2`	$n-f+k+1$	$n+1$	$n+1$	$2n$	$\frac{4}{3}n+\frac{2}{3}$

FIGURE 109: Complete table of delays for variations on "filtering out"

common suffix *if the item occurs in the list.* (This was not the case with other versions based on reversals.) The code can thus be improved so the sharing is maximum in any case. How?

We can compare $\mathcal{W}_n^{\texttt{rm\_lst2}}$ and $\mathcal{W}_n^{\texttt{rm\_lst3}}$ because the worst case of the former happens when the item occurs *anywhere* in the list, whilst the worst case of the latter is when the item occurs uniquely at the end of the list. Indeed, this means that for any input list of length n, $\mathcal{W}_n^{\texttt{rm\_lst3}} < \mathcal{W}_n^{\texttt{rm\_lst2}}$. Since both best cases happen when the item is absent, we conclude that $\mathcal{D}_n^{\texttt{rm\_lst3}} < \mathcal{D}_n^{\texttt{rm\_lst2}}$. Furthermore, we observe that $\mathcal{W}_n^{\texttt{rm\_lst3}} < \mathcal{B}_n^{\texttt{rm\_lst}}$ and $\mathcal{W}_n^{\texttt{rm\_lst3}} < \mathcal{B}_n^{\texttt{rm\_lst\_tf}}$, so $\mathcal{D}_n^{\texttt{rm\_lst3}} < \mathcal{D}_n^{\texttt{rm\_lst}}$ and $\mathcal{D}_n^{\texttt{rm\_lst\_tf}}$. The only way to compare `rm_lst3/2` to `rm_lst1/2` is to consider their exact delays. We have, for all $f \leqslant k$,

$$k < n \Rightarrow \mathcal{D}_{n,f,k}^{\texttt{rm\_lst3}} < \mathcal{D}_{n,f,k}^{\texttt{rm\_lst1}}.$$

Therefore, the fastest function is `rm_lst3/2`.

Note: Using the `case` construct presented in section 12 on page 253, we can write more clearly the code above as

```
rm_lst3(_,   []) -> [];
rm_lst3(I,[I|P]) -> case mem(I,P) of
                      true  -> [I|rm_lst3(I,P)];
                      false -> P
                    end;
rm_lst3(I,[J|P]) -> [J|rm_lst3(I,P)].
mem(_,   [])     -> false;
mem(I,[I|_])     -> true;
mem(I,[_|P])     -> mem(I,P).
```

Question 4. Reading the definition with $n = 0$, that is, a null period, leads to "..., except the items occurring every 0 positions." Therefore, the output ought to be the same as the input. If n is greater than the length of the list, the items at these extra positions are simply ignored because they don't exist. Interpreting the definition for the case of $n < 0$ is inconclusive, so let us decide, for instance, that this case is the same as $n = 0$. No case has been left out in our analysis. We need a counter to record the current position, thus initially set to `1`, and, when the counter matches the period, it is reset to `1`. For the sake of versatility, we shall set the accumulator as first argument, instead of the last:

```
drop(L,P) -> drop([],1,L,P).

drop(A,_,   [],_) $\xrightarrow{\alpha}$ rev(A);
drop(A,P,[_|L],P) $\xrightarrow{\beta}$ drop(A,1,L,P);
```

drop(A,N,[I|L],P) $\xrightarrow{\gamma}$ drop([I|A],N+1,L,P).

The delay occurs when the delay of the call `rev(A)` is maximum, that is, when the accumulator `A` is the longest. This is achieved when the minimum number of items have been dropped (skipped), that is , when clause β is never used. This means that no item is dropped and the output matches the input, so the accumulator at the end is simply the input reversed. The design analysis reminds us that this happens when the period is either negative or greater than the length of the list. Moreover, $\mathcal{D}_n^{\mathsf{drop}} = 1 + n + 1 + (n + 2) = 2n + 4$, because $\mathcal{D}_n^{\mathsf{rev}} = n + 2$.

Question 5. We are given

```
diff([M,N]) -> M - N;
diff([M|L]) -> M - diff(L).
```

This definition only fails on the empty list. Also, the first clause is in tail form but not the second, because the call `diff(L)` is an argument to the operator (-). If we add an integer accumulator, we end in trouble:

```
diff([]) -> list_too_short;
diff( L) -> diff__(L,0).

diff__([M,N],A) -> (M-N)-A;                              % Wrong
diff__([M|L],A) -> diff__(L,M-A).                        % Wrong
```

because subtraction is not associative: $x - (y - z) \neq (x - y) - z$. A more general method is to be preferred then: the accumulator is a list in which the numbers are stored until we can begin subtracting them. This is equivalent to reverse the input and then performing the delayed series of subtractions:

```
diff_tf([]) -> list_too_short;
diff_tf( L) -> rev_diff(L,[]).

rev_diff(   [],A) -> diff__(A,0);          % Prepare to subtract
rev_diff([I|L],A) -> rev_diff(L,[I|A]).                % Reversal

diff__(   [],N) -> N;
diff__([I|L],N) -> diff__(L,I-N).
```

The delay of `diff/1` is $n - 1$, when the input contains n numbers. The delay of `diff_tf/1` is $\mathcal{D}_n^{\mathsf{diff\_tf}} = 1 + (n + 1) + (n + 1) = 2n + 3$.

Question 6. We are given two definitions not in tail form:

```
srev(   []) -> [];
srev([I|L]) -> join(srev(L),[I]).
```

```
join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

We have

```
srev_tf([]) -> [];
srev_tf( L) -> srev__(L,[]).

srev__(   [],[I|A]) -> join__([],[I],[],A);
srev__([I|L],    A) -> srev__(L,[I|A]).

join__(   [],Q,   [],   []) -> Q;
join__(   [],Q,   [],[I|B]) -> join__(Q,[I],[],B);
join__(   [],Q,[I|A],    B) -> join__([],[I|Q],A,B);
join__([I|P],Q,    A,    B) -> join__(P,Q,[I|A],B).
```

Delay and Tail Form Revisited

[See questions on page 156.]

Question 1.

```
split(L,N) when N > 0 -> split([],L,N).

split(A,P=[_|_],0) -> {rev(A),P};
split(A,  [I|L],N) -> split([I|A],L,N-1).

% Reminder
%
rev(L) -> rev_join(L,[]).
rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Question 2.

```
rot(L,N) when N < 0 -> aux(len(L));
rot(L,N)            -> rot([],L,N rem len(L)).

aux(Len) -> rot([],L,Len + N rem Len).

rot([],    L,0) -> L;
rot( A,    L,0) -> join(L,rev(A));
rot( A,[I|L],N) -> rot([I|A],L,N-1).

% Reminder
```

```
%
rev(L) -> rev_join(L,[]).

rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).

join(   [],Q) -> Q;
join([I|P],Q) -> [I|join(P,Q)].
```

Queues

[See questions on page 340.]

Question 1.

```
-module(r).
-export([enqueue/2,dequeue/1,head/1]).

enqueue(I,{[], []}) -> {[],[I]};
enqueue(I,{In,Out}) -> {[I|In],Out}.

dequeue({In,    [I]}) -> {{[],rev(In)},I};
dequeue({In,[I|Out]}) -> {{In,Out},I}.

head({_,[I|_]}) -> I.     % Delay is 1

rev(L) -> rev_join(L,[]).

rev_join(   [],Q) -> Q;
rev_join([I|P],Q) -> rev_join(P,[I|Q]).
```

Question 2. Skipped.

Aliasing and Tail-Call Optimisation

[See question on page 174.]

The only sharing between the input and the output lies in the items of the list. This was expected because when building a list in reverse we cannot reuse any part of the original superstructure.

Persistence and Backtracking

[See questions on page 191.]

Question 1. Skipped.

Question 2. Skipped.

Question 3.

- [`version/1`] Skipped.
- [`set/3`] Either the index is in bounds or not:

  ```
  set(E,I,{{L,U},D,S}) when L =< I,I =< U -> □.
  ```

  Note the conjunction of tests in the guard, meaning here "when `L =< I` *and* `I =< U`." The same effect can be achieved with an auxiliary function, like

  ```
  set(E,I,A={{L,_},_,_}) when L =< I -> aux(E,I,A);
  aux(E,I,  {{L,U},D,S}) when I =< U -> □.
  ```

  The case "empty" is easy because it means that the index is necessary out of bounds, according to the specification. This is the same as having a non-empty array but the index is out of bounds:

  ```
  set(E,I,{{L,U},D,S}) when L =< I,I =< U -> □;
  set(_,_,          _)                    -> out.
  ```

  We can see now that we can remove the first clause:

  ```
  set(E,I,{{L,U},D,S}) when L =< I,I =< U -> □;
  set(_,_,          _)                    -> out.
  ```

  We need now an auxiliary function `set__/3` such that the function call `set__(`E`,`I`,`S`)` rewrites into the the updated S.

  ```
  set(E,I,{B={L,U},D,S}) when L =< I,I =< U ->
                                         {B,D,set__(E,I,S)};
  set(_,_,          _)                    -> out.
  set__(E,I,         {})                  -> □;
  set__(E,I,{set,E,J,S})                  -> □.
  ```

  The first clause of `set__/3` corresponds to an out of bound access:

  ```
  set__(_,_,         {})                  -> out;
  set__(E,I,{set,E,J,S})                  -> □.
  ```

  We realise now that we actually do not need to know the details of the first update because what only matters is that there is at least one. Then we add the assignment:

  ```
  set__(_,_,{})                -> out;
  set__(E,I, S)                -> {set,E,I,S}.
  ```

- [`get/2`] The beginning of the analysis is identical to the analysis of `set/3` until:

```
get({{L,U},_,S},I) when L =< I,I =< U -> get__(S,I);
get(         _,_)                     -> out.
get__(         {},_)                  -> out;
get__({set,E,J,S},I)                  -> □.
```

The remaining clause needs its head to be refined in two cases: either `I` has been found in the current assignment or not:

```
get__(         {},_)                  -> out;
get__({set,E,I,A},I)                  -> □.
get__({set,E,J,A},I)                  -> □.
```

If found, the end with the current content of the cell; otherwise, we need to check the previous assignments, that is, perform a recursive call on the rest of the updates:

```
get__(         {},_)                  -> out;
get__({set,E,I,_},I)                  -> E;
get__({set,_,_,A},I)                  -> get__(A,I).
```

- [`nth/2`] This function can simply be expressed in terms of `get/2`:

```
nth(A={{L,_},_,_},I) -> get(A,L+I-1).
```

- [`mem/2`] Skipped.
- [`inv/1`] Skipped.
- [`unset/2`]

```
unset({B={L,U},D,S},I) when L =< I,I =< U ->
                                           {B,D,unset__(S,I)};
unset__(         {},_) -> out;                         % Absent
unset__({set,_,I,A},I) -> A;                           % Found
unset__({set,E,J,A},I) -> {set,E,J,unset__(A,I)}.
```

Permutations and Sorting

[See question on page 205.] The straight application of our algorithm leads to the following code.

```
-module(p).
-export([perm_tf/1]).

perm_tf(L) -> perm(L,{}).

perm(   [],A) -> appk([],A);
perm(  [I],A) -> appk([[I]],A);
perm([I|L],A) -> perm(L,{k7,I,A}).
```

```
dist(_,      [],A) -> appk([],A);
dist(I,[Perm|P],A) -> dist(I,P,{k5,I,Perm,A}).

insert(I,        [],A) -> appk([[I]],A);
insert(I,Perm=[J|L],A) -> insert(I,L,{k3,[I|Perm],J,A}).

push(_,   [],A) -> appk([],A);
push(I,[L|H],A) -> push(I,H,{k2,[I|L],A}).

join(   [],Q,A) -> appk(Q,A);
join([I|P],Q,A) -> join(P,Q,{k1,I,A}).

appk(V,     {k7,I,A}) -> dist(I,V,A);
appk(V,     {k6,W,A}) -> join(V,W,A);
appk(V,{k5,I,Perm,A}) -> insert(I,Perm,{k6,V,A});
appk(V,     {k4,P,A}) -> appk([P|V],A);
appk(V,   {k3,P,J,A}) -> push(J,V,{k4,P,A});
appk(V,     {k2,P,A}) -> appk([P|V],A);
appk(V,     {k1,I,A}) -> appk([I|V],A);
appk(V,           {}) -> V.
```

The clauses of `appk/2` matching the tags `k1`, `k2` and `k4` have isomorphic bodies, so they can be merged into one clause. To keep track of the fusion, we call the new tag `k124` in the following last version.

```
-module(q).
-export([perm_tf/1]).

perm_tf(L) -> perm(L,{}).

perm(   [],A) -> appk([],A);
perm(  [I],A) -> appk([[I]],A);
perm([I|L],A) -> perm(L,{k7,I,A}).

dist(_,      [],A) -> appk([],A);
dist(I,[Perm|P],A) -> dist(I,P,{k5,I,Perm,A}).

insert(I,        [],A) -> appk([[I]],A);
insert(I,Perm=[J|L],A) -> insert(I,L,{k3,[I|Perm],J,A}).

push(_,   [],A) -> appk([],A);
push(I,[L|H],A) -> push(I,H,{k124,[I|L],A}).
```

```
join(   [],Q,A) -> appk(Q,A);
join([I|P],Q,A) -> join(P,Q,{k124,I,A}).

appk(V,     {k7,I,A}) -> dist(I,V,A);
appk(V,     {k6,W,A}) -> join(V,W,A);
appk(V,{k5,I,Perm,A}) -> insert(I,Perm,{k6,V,A});
appk(V,   {k3,P,J,A}) -> push(J,V,{k124,P,A});
appk(V,   {k124,I,A}) -> appk([I|V],A);
appk(V,           {}) -> V.
```

Insertion Sort

[See questions on page 240.]

Question 1. Let us recall the code and label the arrows as follows:

```
i2w_a(L) -> i2w_a([],[],L).

i2w_a(   [],    Q,     []) $\xrightarrow{\alpha}$ Q;
i2w_a([I|P],    Q,     []) $\xrightarrow{\beta}$ i2w_a(    P,[I|Q],[]);
i2w_a(    P,[J|Q],L=[K|_]) when J < K
                           $\xrightarrow{\gamma}$ i2w_a([J|P],    Q, L);
i2w_a(   [],    Q,  [K|R]) $\xrightarrow{\delta}$ i2w_a(  [K],    Q, R);
i2w_a([I|P],    Q,L=[K|_]) when K < I
                           $\xrightarrow{\epsilon}$ i2w_a(    P,[I|Q], L);
i2w_a(    P,    Q,  [K|R]) $\xrightarrow{\zeta}$ i2w_a(    P,[K|Q], R).
```

Clause δ (in bold) leads to a slightly more balanced scheme than `isort2w/1`, insofar as when the left list is empty, the item under consideration will be pushed in it instead of the right list. Let us start with a small example, shown in FIGURE. 110 on page 383. Note how the configuration `[]`,`[`c`,`a`,`b`]` is not reached anymore, as it is maximally unbalanced. Therefore, we should expect in average a little improvement over the original function `isort2w/1`. We shall skip the best and worst cases and instead focus on the average delay as follows. Let us note $\mathcal{A}_{p,q}$ the average delay for inserting one number in a configuration where the left list is made of p numbers and the right list contains q numbers, all of them being unique. This function was on page 228:

$$\mathcal{A}^{\gamma\delta\epsilon}_{k-q,q} = \frac{1}{k+1}q^2 - \frac{k}{k+1}q + \frac{k+2}{2}.$$

Let $\mathcal{A}_n$ represent the average delay required to insert a random number into two random lists of n numbers in total. This function is different. As hinted, the difference with the analysis of `isort2w/1` consists in *not* having to sum the term $\mathcal{A}_{0,k}$ to compute $\mathcal{A}_n$ and, in its stead,

$$\begin{tikzcd}
 & & & {[a],[\,]} \arrow[dll, "\epsilon"'] \arrow[dr, "\zeta"] & & \\
 & {[\,],[a]} \arrow[d, "\delta"'] & & & {[a],[b]} \arrow[dl, "\gamma"'] \arrow[d, "\epsilon"] \arrow[dr, "\zeta"] & \\
 & {[b],[a]} \arrow[dl, "\gamma"'] \arrow[d, "\epsilon"] \arrow[dr, "\zeta"] & & {[b,a],[\,]} \arrow[d, "\zeta"'] & {[\,],[a,b]} \arrow[d, "\delta"'] & {\begin{array}{c}[a],[c,b]\\(1,3,2)\end{array}} \\
{[a,b],[\,]} \arrow[d, "\zeta"'] & {[\,],[b,a]} \arrow[d, "\delta"'] & {\begin{array}{c}[b],[c,a]\\(2,3,1)\end{array}} & {\begin{array}{c}[b,a],[c]\\(2,1,3)\end{array}} & {\begin{array}{c}[c],[a,b]\\(3,1,2)\end{array}} & \\
{\begin{array}{c}[a,b],[c]\\(1,2,3)\end{array}} & {\begin{array}{c}[c],[b,a]\\(3,2,1)\end{array}} & & & &
\end{tikzcd}$$

FIGURE 110: Slightly balanced two-way insertions with `i2w_a/1`

use $\mathcal{A}_{1,k-1}$, this change corresponding to the added clause. We have

$$\mathcal{A}_0 := 1, \quad \mathcal{A}_1 := \frac{3}{2},$$
$$\mathcal{A}_k := \frac{1}{k}\left(\sum_{p+q=k}^{p,q>0}\mathcal{A}_{p,q} + \mathcal{A}_{1,k-1}\right) = \frac{1}{k}\left(\sum_{q=1}^{k-1}\mathcal{A}_{k-q,q} + \mathcal{A}_{1,k-1}\right).$$

The values of $\mathcal{A}_0$ and $\mathcal{A}_1$ come from examining FIGURE 110 on page 383. Because

$$\mathcal{A}_{1,k-1} = \frac{k^2+k+4}{2k+2},$$

we resume

$$\mathcal{A}_k = \frac{1}{k}\left(\frac{1}{k+1}\sum_{q=1}^{k-1}q^2 - \frac{k}{k+1}\sum_{q=1}^{k-1}q + \frac{k+2}{2}\sum_{q=1}^{k-1}1 + \frac{k^2+k+4}{2k+2}\right)$$
$$= \frac{2k^3+9k^2+k+6}{6k(k+1)} = \frac{k}{3} + \frac{1}{k} - \frac{2}{k+1} + \frac{7}{6}.$$

Let $\mathcal{A}_k^{\alpha\beta}$ be the number of rewrites to reverse k numbers from the first list to the second and finally return the second list. We have

$$\mathcal{A}_k^{\alpha\beta} := k+1; \quad \mathcal{A}_0^{\texttt{i2w\_a}} := 1,$$
$$\mathcal{A}_n^{\texttt{i2w\_a}} := 1 + \sum_{k=0}^{n-1}\mathcal{A}_k + \frac{1}{n}\left(\sum_{k=1}^{n-1}\mathcal{A}_k^{\alpha\beta} + 2\right)$$

$$
\begin{aligned}
&= 1 + \left(\mathcal{A}_0 + \mathcal{A}_1 + \sum_{k=2}^{n-1}\mathcal{A}_k\right) + \frac{1}{n}\left(\sum_{k=1}^{n-1}(k+1) + 2\right)\\
&= \frac{7}{2} + \sum_{k=2}^{n-1}\left(\frac{k}{3} + \frac{1}{k} - \frac{2}{k+1} + \frac{7}{6}\right) + \frac{1}{n}\left(\sum_{k=1}^{n} k - 1\right) + \frac{2}{n}\\
&= \frac{7}{2} + \sum_{k=1}^{n-1}\left(\frac{k}{3} + \frac{1}{k} - \frac{2}{k+1} + \frac{7}{6}\right) - \left(\frac{1}{3} + \frac{1}{1} - \frac{2}{1+1} + \frac{7}{6}\right)\\
&\quad + \frac{1}{n}\cdot\frac{n(n+1)}{2} - \frac{1}{n} + \frac{2}{n}\\
&= \left(\frac{1}{3}\sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1}\frac{1}{k} - 2\sum_{k=1}^{n-1}\frac{1}{k+1} + \frac{7}{6}(n-1)\right) + \frac{n}{2} + \frac{1}{n} + \frac{5}{2}\\
&= \frac{1}{3}\cdot\frac{n(n-1)}{2} + \sum_{k=1}^{n}\frac{1}{k} - 2\sum_{k=2}^{n}\frac{1}{k} + \frac{5}{3}n + \frac{4}{3}\\
&= \frac{1}{6}n^2 + \frac{3}{2}n + H_n - 2(H_n - 1) + \frac{4}{3} = \frac{1}{6}n^2 + \frac{3}{2}n - H_n + \frac{10}{3}.
\end{aligned}
$$

Furthermore,

$$\mathcal{A}_n^{\texttt{i2w\_a}} = \frac{1}{6}n^2 + \frac{3}{2}n - H_n + \frac{10}{3} < \frac{1}{6}n^2 + \frac{3}{2}n + \frac{4}{3} = \mathcal{A}_n^{\texttt{isort2w}} \Leftrightarrow 2 < H_n.$$

Moreover, since $(H_n)_n$ is increasing and $H_3 = 11/6$ and $H_4 = 25/12$,

$$H_3 < 2 < H_4 \leqslant H_n,$$

so $\mathcal{A}_n^{\texttt{i2w\_a}} < \mathcal{A}_n^{\texttt{isort2w}} \Leftrightarrow 3 < n$. This means that if four or more items are to be sorted, it is faster to call `i2w_a/1` than `isort2w/1`.

Higher-Order Functions

[See questions on page 282.]

```
-module(ho).
-export([max_f/3,max_x/3]).

% Question 1
%
% The test A > B ensures that A and B are integers
%
max_int(A,B) when A =< B -> B;
max_int(A,B) when A > B  -> A.

max_f(F,B,B)            -> F(B);
max_f(F,A,B) when A < B -> max_int(F(A),max_f(F,A+1,B)).
```

```
% Question 2
%
max_x(_,B,B) -> B;
max_x(F,A,B) when A < B -> M = max_x(F,A+1,B),
                           case F(A) >= M of
                             true  -> A;
                             false -> M
                           end.
```

Merge Sort

[See questions on page 326.]

Question 1. It is not necessary to redo all the calculations. The shortcut reveals itself if we start with assessing the difference between $\overline{\mathcal{D}}^{\mathsf{merge}}_{m,n}$, which is the number of comparisons, and $\mathcal{D}^{\mathsf{merge}}_{m,n}$, which is the number of function calls. Let us recall the Erlang definition of `merge/2`:

```
merge(   [],      Q)             -α-> Q;
merge(    P,     [])             -β-> P;
merge([I|P],Q=[J|_]) when I < J  -γ-> [I|merge(P,Q)];
merge(    P,  [J|Q])             -δ-> [J|merge(P,Q)].
```

In order to only count the number of comparisons, we must not count clauses α and β in the delay: $\overline{\mathcal{D}}^{\mathsf{merge}}_{m,n} = \mathcal{D}^{\mathsf{merge}}_{m,n} - 1$. Thus

$$\begin{aligned}\overline{\mathcal{B}}^{\mathsf{merge}}_{m,n} &= \mathcal{B}^{\mathsf{merge}}_{m,n} - 1 = \min\{m,n\};\\ \overline{\mathcal{W}}^{\mathsf{merge}}_{m,m} &= \mathcal{W}^{\mathsf{merge}}_{m,n} - 1 = m+n-1;\\ \overline{\mathcal{A}}^{\mathsf{merge}}_{m,m} &= \mathcal{A}^{\mathsf{merge}}_{m,n} - 1 = mn/(m+1) + mn/(n+1).\end{aligned}$$

The shape of equation (13.27) on page 296 is the same, except it applies now to the number of comparisons:

$$\begin{aligned}\overline{\mathcal{D}}(2^p) &= 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}\overline{\mathcal{D}}^{\mathsf{merge}}_{2^k,2^k} = 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}\left(\mathcal{D}^{\mathsf{merge}}_{2^k,2^k} - 1\right)\\ &= 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k}\mathcal{D}^{\mathsf{merge}}_{2^k,2^k} - 2^{p-1}\sum_{k=0}^{p-1}\frac{1}{2^k} = \mathcal{D}(2^p) - 2^{p-1}\left(2 - \frac{1}{2^{p-1}}\right)\\ &= \mathcal{D}(2^p) - 2^p + 1.\end{aligned}$$

Therefore

$$\overline{\mathcal{B}}(2^p) = \mathcal{B}(2^p) - 2^p + 1 = p2^{p-1};$$

$$\overline{\mathcal{W}}(2^p) = \mathcal{W}(2^p) - 2^p + 1 = p2^p - 2^p + 1;$$
$$\overline{\mathcal{A}}(2^p) = \mathcal{A}(2^p) - 2^p + 1 = p2^p - 2^p \sum_{k=0}^{p-1} \frac{1}{2^k + 1}.$$

For the general case, let us reconsider equation (13.36) on page 302:

$$\mathcal{D}(n) = \sum_{i=0}^{r} \mathcal{D}(2^{e_i}) + \sum_{i=1}^{r} \mathcal{D}^{\texttt{merge}}_{2^{e_i}, 2^{e_{i-1}} + \cdots + 2^{e_0}}.$$

Let us take into account only the number of comparisons:

$$\begin{aligned}
\overline{\mathcal{D}}(n) &= \sum_{i=0}^{r} \overline{\mathcal{D}}(2^{e_i}) + \sum_{i=1}^{r} \overline{\mathcal{D}}^{\texttt{merge}}_{2^{e_i}, 2^{e_{i-1}} + \cdots + 2^{e_0}} \\
&= \sum_{i=0}^{r} (\mathcal{D}(2^{e_i}) - 2^{e_i} + 1) + \sum_{i=1}^{r} \left(\mathcal{D}^{\texttt{merge}}_{2^{e_i}, 2^{e_{i-1}} + \cdots + 2^{e_0}} - 1\right) \\
&= \sum_{i=0}^{r} \mathcal{D}(2^{e_i}) - \sum_{i=0}^{r} 2^{e_i} + (r+1) + \sum_{i=1}^{r} \mathcal{D}^{\texttt{merge}}_{2^{e_i}, 2^{e_{i-1}} + \cdots + 2^{e_0}} - r, \\
\overline{\mathcal{D}}(n) &= \mathcal{D}(n) - n + 1.
\end{aligned}$$

It is easy to derive the best, worst and average number of comparisons:

$$\begin{aligned}
\tfrac{1}{2} n \lg n - \tfrac{3}{2} n + 1 &\leqslant \overline{\mathcal{B}}(n) \leqslant \tfrac{1}{2} n \lg n + n - 1, \\
n \lg n - 4n + 3 &\leqslant \overline{\mathcal{W}}(n) \leqslant n \lg n + n - 1, \\
n \lg n - (\alpha + 3)n + \tfrac{3}{2} &\leqslant \overline{\mathcal{A}}(n) \leqslant n \lg n + n - 1.
\end{aligned}$$

We have $S(n) \leqslant \overline{\mathcal{W}}(n)$, therefore, $S(n) \leqslant n \lg n + n - 1$.

Question 2. If $n = 2^p - 1$, then the binary representation of n is only made of 1-bits. In other words, if we use the variant representation $n = 2^{e_r} + \cdots + 2^{e_1} + 2^{e_0} > 0$, with $e_r > \cdots > e_1 > e_0 \geqslant 0$ and $r \geqslant 0$, it means that $e_i = i$. Let us start by recalling the delays when $n = 2^p$. Equality (13.29) on page 297 is: $\mathcal{B}(2^p) = p2^{p-1} + 2^p - 1$. Equality (13.30) on page 297: $\mathcal{W}(2^p) = p2^p$. Reusing equation (13.38) on page 303, defining $\mathcal{B}(n)$ in the general, with the additional constraint $e_i = i$, becomes

$$\begin{aligned}
\mathcal{B}(n) &= \sum_{i=0}^{r} \mathcal{B}(2^{e_i}) + r + \sum_{i=1}^{r} \sum_{j=0}^{i-1} 2^{e_j} = \sum_{i=0}^{r} \mathcal{B}(2^i) + r + \sum_{i=1}^{r} \sum_{j=0}^{i-1} 2^j \\
&= \sum_{i=0}^{r} (i2^{i-1} + 2^i - 1) + r + \sum_{i=1}^{r} (2^i - 1) \\
&= \sum_{i=1}^{r} i2^{i-1} + 2^{r+2} - r - 4.
\end{aligned}$$

A closed form for the remaining sum can be reached by the *perturbation technique*, which consists in summing one more term and rewriting the new sum in terms of the original sum. Let $S_r := \sum_{i=1}^{r} i2^{i-1}$. Then

$$\begin{aligned}S_r + (r+1)2^r &= \sum_{i=1}^{r+1} i2^{i-1} = \sum_{i=0}^{r} (i+1)2^i = \sum_{i=0}^{r} i2^i + \sum_{i=0}^{r} 2^i\\ &= 2\sum_{i=1}^{r} i2^{i-1} + (2^{r+1}-1) = 2S_r + 2^{r+1} - 1,\end{aligned}$$

yielding $S_r = (r+1)2^r - 2^{r+1} + 1 = (r-1)2^r + 1$. Then

$$\mathcal{B}(n) = ((r-1)2^r + 1) + 2^{r+2} - r - 4 = (r+3)(2^r - 1).$$

We assumed $e_i = i$, that is, the bits of n are all 1, which implies that there exists p such that $n = 2^p - 1$, hence $\nu_n := r + 1 = p$. We conclude:

$$\mathcal{B}(2^p - 1) = (p+2)(2^{p-1} - 1).$$

We may remark that $\mathcal{B}(2^p) = \mathcal{B}(2^p - 1) + p + 1$. Furthermore, reusing the result $\overline{\mathcal{B}}(n) = \mathcal{B}(n) - n + 1$ of Question 1, we would deduce

$$\overline{\mathcal{B}}(2^p) = \overline{\mathcal{B}}(2^p - 1) + p. \tag{16.67}$$

Let us turn now to equation (13.48) on page 306 defining $\mathcal{W}(n)$:

$$\begin{aligned}\mathcal{W}(n) &= \sum_{i=0}^{r} \mathcal{W}(2^{e_i}) + \sum_{i=1}^{r}\sum_{j=0}^{i} 2^{e_j} = \sum_{i=0}^{r} e_i 2^{e_i} + \sum_{i=1}^{r}\sum_{j=0}^{i} 2^{e_j}\\ &= \sum_{i=0}^{r} i2^i + \sum_{i=1}^{r}\sum_{j=0}^{i} 2^j = 2S_r + \sum_{i=1}^{r}(2^{i+1} - 1),\\ \mathcal{W}(2^p - 1) &= (r+1)2^{r+1} - r - 2 = p2^p - p - 1.\end{aligned}$$

Remarkably, we stumble upon a familiar pattern again:

$$\overline{\mathcal{W}}(2^p) = \overline{\mathcal{W}}(2^p - 1) + p. \tag{16.68}$$

Question 3. Let us take $a = 1$, $b = n$ and $f(x) = \lg x$. The theorem implies

$$\int_1^n \lg x \, dx \leqslant \sum_{k=2}^{n} \lg k = \lg n! \leqslant S(n).$$

Since

$$\int_1^n \lg x \, dx = \frac{1}{\ln 2}[x \ln x - x]_1^n = n \lg n - \frac{n}{\ln 2} + \frac{1}{\ln 2},$$

we conclude, together with the result of Question 1:

$$n \lg n - \frac{n}{\ln 2} + \frac{1}{\ln 2} \leqslant S(n) \leqslant n \lg n + n - 1.$$

Question 4. We found earlier $n\lfloor\lg n\rfloor/2 + 2n - 2^{\lfloor\lg n\rfloor+1} \leqslant \mathcal{B}(n)$. We want to find the largest λ' such that

$$\tfrac{1}{2}n\lg n + \lambda' n \leqslant \tfrac{1}{2}n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+1}.$$

Equivalently, $n\lg n + 2\lambda' n \leqslant n\lfloor\lg n\rfloor + 4n - 2^{\lfloor\lg n\rfloor+2}$, which is an instance of $n\lg n + \lambda n \leqslant n\lfloor\lg n\rfloor + \psi n - \omega 2^{\lfloor\lg n\rfloor}$, with $\lambda = 2\lambda'$, $\psi = 4$ and $\omega = 4$. Therefore, let us work out the general case where ψ and ω are positive integers. We set $n = q2^p$, where $p \in \mathbb{N}, q \in \mathbb{R}$ and $1 \leqslant q < 2$. Thus $\lfloor\lg n\rfloor = p$ and the previous inequality is equivalent to $q2^p(\lg q + p) + \lambda q 2^p \leqslant pq2^p + \psi q 2^p - \omega 2^p$, which, in turn, is equivalent to $q\lg q + (\lambda - \psi)q + \omega \leqslant 0$. Let us define $f(q) := q\lg q + (\lambda-\psi)q + \omega$. We have $f'(q) = \lg q + 1/\ln 2 + \lambda - \psi$ and $f'(q) = 0 \Leftrightarrow q = 1/e/2^{\lambda-\psi}$. Furthermore, $\lim_{q\rightarrow 0^+} f(q) = \omega^-$ and $\lim_{q\rightarrow+\infty} f(q) = +\infty$. This means that f can be continuously extended by setting $f(0) := \lambda - \psi$, that it decreases to its minimum $f(1/e/2^{\lambda-\psi})$ and then increases without a bound. Let us allow $q = 2$ and consider what happens at the bounds of the interval $[1,2]$: $f(1) \leqslant 0 \Leftrightarrow \lambda - \psi \leqslant -\omega$ and $f(2) \leqslant 0 \Leftrightarrow \lambda - \psi \leqslant -\omega/2 - 1$. Either $-\omega \leqslant -\omega/2 - 1$ or else the converse holds. In other words, either $\omega \geqslant 2$ or $\omega \leqslant 2$. Since the bounds we are interested in have $\omega \geqslant 2$, we shall only consider the former case, from which we deduce that $f(1) \leqslant 0 \Rightarrow f(2) \leqslant 0$. Let us then set $\lambda - \psi := -\omega$ and prove that $f(q) \leqslant 0$ then holds for $q \in [1,2]$. We have now $f(q) = q\lg q - \omega q + \omega = q\lg q + (1-q)\omega \leqslant 0 \Leftrightarrow q\lg q \leqslant (q-1)\omega$. Since $1 \leqslant q$, this is equivalent to $q\lg q/(q-1) \leqslant \omega$. Let us define $g(q) := q\lg q/(q-1)$. Then $g'(q) = (-(\ln 2)\lg q + q - 1)/(q-1)^2/\ln 2$. The denominator is positive, so we turn our attention to the numerator by setting $h(q) := -(\ln 2)\lg q + q - 1$, yielding the derivative $h'(q) = 1 - 1/q$, which is positive because $q \geqslant 1$. So h increases and, since $h(1) = 0$, we have $g'(q) \geqslant 0$ for $q \geqslant 1$. Thus g increases and reaches its maximum in $[1,2]$ at the upper bound: $g(2) = 2$. Since we set $\omega \geqslant 2$, we proved that $f(q) \leqslant 0$, for all $q \in [1,2]$. Because we defined $\lambda - \psi := -\omega$ and $\lambda = 2\lambda'$, we have the answer $\lambda' = (\psi - \omega)/2$. In the example above, $\psi = 4$ and $\omega = 4$, we can check that $\omega \geqslant 2$ and thus $\lambda' = 0$. In other words:

$$\tfrac{1}{2}n\lg n \leqslant \tfrac{1}{2}n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+1} \leqslant \mathcal{B}(n).$$

Let us apply the same theorem to the following remaining lower bounds. First, $n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+2} + 2 \leqslant \mathcal{W}(n)$. Inequation $n\lg n + \lambda n + 2 \leqslant n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+2} + 2$ is equivalent to $n\lg n + \lambda n \leqslant n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+2}$, for which we want to maximise λ. We have $\psi = 2$ and $\omega = 4 \geqslant 2$. Thus $\lambda = \psi - \omega = -2$ and

$$n\lg n - 2n + 2 \leqslant n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+2} + 2 \leqslant \mathcal{W}(n).$$

Finally, $n\lfloor\lg n\rfloor + (3-\alpha)n - 2^{\lfloor\lg n\rfloor+2} + 1/2 \leqslant \mathcal{A}(n)$. Inequation $n\lg n + \lambda n + 1/2 \leqslant n\lfloor\lg n\rfloor + (3-\alpha)n - 2^{\lfloor\lg n\rfloor+2} + 1/2$ is equivalent to $n\lg n + \lambda n \leqslant n\lfloor\lg n\rfloor + (3-\alpha)n - 2^{\lfloor\lg n\rfloor+2}$, for which we want to maximise λ. We have $\psi = 3-\alpha$ and $\omega = 4 \geqslant 2$. Thus $\lambda = \psi - \omega = -1-\alpha$ and

$$n\lg n - (1+\alpha)n + 1/2 \leqslant n\lfloor\lg n\rfloor + (3-\alpha)n - 2^{\lfloor\lg n\rfloor+2} + 1/2 \leqslant \mathcal{A}(n).$$

Let us now modify our previous line of argumentation to tackle upper bounds. In other words, given ψ and ω, we want to find the *minimum* λ such that $n\lfloor\lg n\rfloor + \psi n - \omega 2^{\lfloor\lg n\rfloor} \leqslant n\lg n + \lambda n$. We derive $f(q) \geqslant 0$, with $f(q) := q\lg q + (\lambda-\psi)q + \omega$. From $\omega \geqslant 2$ we deduce $f(2) \geqslant 0 \Rightarrow f(1) \geqslant 0$. Let us set $\lambda - \psi := -\omega/2 - 1$ and prove that $f(q) \geqslant 0$ for $q \in [1,2]$. We have now $f(q) = q\lg q - (\omega/2+1)q + \omega \geqslant 0 \Leftrightarrow (2-q)\omega \geqslant 2q(1-\lg q)$. Since $q \leqslant 2$, this is equivalent to $\omega \geqslant 2q(\lg q - 1)/(q-2)$. Let $g(q) := q(\lg q - 1)/(q-2)$. Then $g'(q) = (-(2\ln 2)\lg q + q + 2\ln 2 - 2)/(q-2)^2/\ln 2$. The denominator is positive, so let us focus on determining the sign of the numerator by setting $h(q) := -(2\ln 2)\lg q + q + 2\ln 2 - 2$, yielding the derivative $h'(q) = 1 - 2/q$, and $q \leqslant 2 \Rightarrow h'(q) \leqslant 0$. Therefore, h decreases and, because $h(1) = 2\ln 2 - 1 > 0$ and $h(2) = 0$, we conclude that $h(q) \geqslant 0$ and then $g'(q) \geqslant 0$, which means that $g(q)$ is increasing and reaches its maximum at the limit $q = 2$. This limit can be found by applying *L'Hôpital's Rule*: $\lim_{q\to 2} g(q) = 1/\ln 2$. Therefore, $2.88 \simeq 2/\ln 2 \geqslant 2q(\lg q - 1)/(q-2)$, when $q \in [1,2]$ and the bound is tight when $q \to 2$. If we restrict the range of ω so $\omega \geqslant 3$ (it is an integer), then we prove $f(q) \geqslant 0$, for all $q \in [1,2]$ and $\lambda = \psi - \omega/2 - 1$. Consider again $\mathcal{B}(n) \leqslant \frac{1}{2}n\lfloor\lg n\rfloor + 3n - 2^{\lfloor\lg n\rfloor+1} - 1$. This is equivalent to $2\cdot\mathcal{B}(n) \leqslant n\lfloor\lg n\rfloor + 6n - 4\cdot 2^{\lfloor\lg n\rfloor} - 2$, which fits the conditions of the previous theorem, where $\psi = 6$ and $\omega = 4 \geqslant 3$. Then $\lambda = 3$ and

$$\mathcal{B}(n) \leqslant \tfrac{1}{2}n\lg n + \tfrac{3}{2}n - 1.$$

The case $\omega = 2$ is problematic and requires a solution because it occurs in the upper bound of the worst and average delays (it is the same bound): $\mathcal{W}(n) \leqslant n\lfloor\lg n\rfloor + 3n - 2^{\lfloor\lg n\rfloor+1} - 1$. A simple workaround consists in slightly weakening the bound so $\omega \geqslant 3$: $-2^{\lfloor\lg n\rfloor+1} = -3\cdot 2^{\lfloor\lg n\rfloor} + 2^{\lfloor\lg n\rfloor} \leqslant -3\cdot 2^{\lfloor\lg n\rfloor} + n$. In other words, whenever $\omega = 2$, we change ψ into $\psi+1$ and ω into $\omega+1$, then apply the theorem, so $\lambda = \psi - 3/2$. Then $\mathcal{W}(n) \leqslant n\lfloor\lg n\rfloor + 4n - 3\cdot 2^{\lfloor\lg n\rfloor} - 1$ and

$$\mathcal{W}(n) \leqslant n\lg n + \tfrac{3}{2}n - 1.$$

From Question 1, we know that $\overline{\mathcal{W}}(n) = \mathcal{W}(n) - n + 1$, therefore $\overline{\mathcal{W}}(n) \leqslant n\lg n + \frac{1}{2}n$. Furthermore, we have $S(n) \leqslant \overline{\mathcal{W}}(n)$, hence, together with the lower bound from Question 3, we conclude

$$n\lg n - \frac{n}{\ln 2} + \frac{1}{\ln 2} \leqslant S(n) \leqslant n\lg n + \frac{1}{2}n.$$

Question 5. From definition (13.41) on page 303 of the bit sum ν_n,

$$\nu_0 = 0, \quad \nu_{2n} = \nu_n, \quad \nu_{2n+1} = \nu_n + 1,$$

we derive a straightforward implementation in Erlang as follows:

```
nu(0) -> 0;
nu(N) -> nu(N div 2) + N rem 2.
```

where `N div 2` realises $\lfloor n/2 \rfloor$ and `N rem 2` realises $n - 2\lfloor n/2 \rfloor$. (By the way, we have $\mathcal{D}_n^{\mathsf{nu}} = \mathcal{D}_n^{\mathsf{exp2}}$. See definition (13.58) on page 319.)

Question 6. First, let us explicitly express the dependence on n by defining $e_0 := \rho_n$, where ρ_n is *the number of trailing zeros* of n and ρ is called *the ruler function*. Next, we define ρ recursively as

$$\rho_0 := \infty, \qquad \rho_{2n} := \rho_n + 1, \qquad \rho_{2n+1} := 0.$$

Finally, we recall the recursive definition of ν (13.41) on page 303:

$$\nu_0 := 0, \qquad \nu_{2n} := \nu_n, \qquad \nu_{2n+1} := \nu_n + 1.$$

At first sight, a simple relationship between these two functions is not obvious. Without loss in generality, we can assume that $2n + 1 := (\alpha 0 1^a)_2$, where α is an arbitrary bit string and 1^a is a 1-bit string of length a. Then $2n + 2 = (\alpha 1 0^a)_2$ and

$$\nu_{2n+1} = \nu_\alpha + a, \quad \rho_{2n+1} = 0, \quad \nu_{2n+2} = \nu_\alpha + 1, \quad \rho_{2n+2} = a.$$

Now, we can relate ρ and ν by means of a:

$$\rho_{2n+2} = \nu_{2n+1} - \nu_\alpha = \nu_{2n+1} - (\nu_{2n+2} - 1) = 1 + \nu_{2n+1} - \nu_{2n+2}.$$

We can check now that the same pattern also works for ρ_{2n+1} by simply using the definitions of ρ and ν: $\rho_{2n+1} = 1 + \nu_{2n} - \nu_{2n+1}$. This achieves to establish, for any integer $n > 0$, that $e_0 := \rho_n = 1 + \nu_{n-1} - \nu_n$. Summing on both sides leads to $\sum_{k=1}^{n} \rho_k = n - \nu_n$.

Question 7. There are two cases: either $n + 1$ is odd or it is even. If the former, we need to draw the trees corresponding to those in FIGURE 88 on page 328, to be found in FIGURE 111 on page 391; if the latter, we draw trees from those in FIGURE 89 on page 329, as seen in FIGURE 112 on the facing page. Next, we compare all the best delays involved in the trees in FIGURE 88b on page 328, whose sum of the nodes is $\overline{\mathcal{B}}(n)$, and FIGURE 111b on the facing page, whose sum of the nodes is $\overline{\mathcal{B}}(n + 1)$, when n is even. We can clearly see that the latter tree contains an excess of $r + 1 + \overline{\mathcal{B}}(2^0) = r + 1 := \nu_n$. This means that we have $\overline{\mathcal{B}}(n + 1) = \overline{\mathcal{B}}(n) + \nu_n$, when n is even. We need to apply the same scrutiny to the complementary case, n odd, which corresponds to the trees in FIGURE 89b on page 329 and FIGURE 112b on page 391.

$$\begin{tikzcd}
& \sum_{i=0}^{r} 2^{e_i}+1 & & & & \\
2^{e_r} \arrow[ur] & & \sum_{i=0}^{r-1} 2^{e_i}+1 \arrow[ul] & & & \\
& 2^{e_{r-1}} \arrow[ur] & & 2^{e_1}+2^{e_0}+1 \arrow[ul, dotted] & & \\
& & 2^{e_1} \arrow[ur] & & 2^{e_0}+1 \arrow[ul] & \\
& & & 2^{e_0>0} \arrow[ur] & & 2^0 \arrow[ul]
\end{tikzcd}$$

(a) Nodes with list lengths

$$\begin{tikzcd}
& \sum_{i=0}^{r-1} 2^{e_i}+1 & & & & \\
\overline{\mathcal{B}}(2^{e_r}) \arrow[ur] & & \sum_{i=0}^{r-2} 2^{e_i}+1 \arrow[ul] & & & \\
& \overline{\mathcal{B}}(2^{e_{r-1}}) \arrow[ur] & & 2^{e_0}+1 \arrow[ul, dotted] & & \\
& & \overline{\mathcal{B}}(2^{e_1}) \arrow[ur] & & 1 \arrow[ul] & \\
& & & \overline{\mathcal{B}}(2^{e_0}) \arrow[ur] & & \overline{\mathcal{B}}(2^0) \arrow[ul]
\end{tikzcd}$$

(b) Nodes with best delays

FIGURE 111: Best case for $n+1$ odd (see FIGURE 88 on page 328)

The latter contains an excess of $\overline{\mathcal{B}}(2^q)$ (we do not find this node-delay in the other tree) and a default of $\sum_{i=0}^{q-1} \overline{\mathcal{B}}(2^i) - r + \sum_{j=1}^{q-1} 2^j$. That is, when n is odd:

$$\overline{\mathcal{B}}(n+1) - \overline{\mathcal{B}}(2^q) + \sum_{i=0}^{q-1} \overline{\mathcal{B}}(2^i) - r + \sum_{j=1}^{q-1} 2^j = \overline{\mathcal{B}}(n).$$

Because we already found that $\overline{\mathcal{B}}(2^p) = p2^{p-1}$, this is equivalent to

$$\overline{\mathcal{B}}(n+1) = \overline{\mathcal{B}}(n) + q2^{q-1} - \sum_{i=0}^{q-1} i2^{i-1} + r - (2^q - 2).$$

From Question 2, we know that $\sum_{i=0}^{r} i2^{i-1} = (r-1)2^r + 1$, so

$$\begin{aligned}\overline{\mathcal{B}}(n+1) &= \overline{\mathcal{B}}(n) + q2^{q-1} - (q-2)2^{q-1} - 1 + r - 2^q + 2\\ &= \overline{\mathcal{B}}(n) + r + 1,\end{aligned}$$

that is, $\overline{\mathcal{B}}(n+1) = \overline{\mathcal{B}}(n) + \nu_n$, just as in the case n even. Therefore, the equality is independent of the parity of n. By summing both sides

$$\begin{tikzcd}
& \sum_{i=q}^{r} 2^{e_i}+2^q & & \\
2^{e_r} \arrow[ur] & & 2^{e_q}+2^q \arrow[ul, dotted] & \\
& 2^{e_q} \arrow[ur] & & 2^q \arrow[ul]
\end{tikzcd}$$

(a) Nodes with list lengths

$$\begin{tikzcd}
& \sum_{i=q}^{r-1} 2^{e_i}+2^q & & \\
\overline{\mathcal{B}}(2^{e_r}) \arrow[ur] & & 2^q \arrow[ul, dotted] & \\
& \overline{\mathcal{B}}(2^{e_q}) \arrow[ur] & & \overline{\mathcal{B}}(2^q) \arrow[ul]
\end{tikzcd}$$

(b) Nodes with best delays

FIGURE 112: Best case for $n+1$ even (see FIGURE 89 on page 329)

from 0 to $n-1$, we obtain the pretty *exact* formula

$$\overline{\mathcal{B}}(n) = \overline{\mathcal{B}}(0) + \sum_{i=0}^{n-1} \nu_i = \sum_{i=0}^{n-1} \nu_i.$$

Let us look at FIGURE 87 on page 326. Note that $n = 0$ is missing in the table, but it is important to consider it here and also to fill the blanks in the table with 0s. We can see bit strings whose pattern is $0^{2^i}1^{2^i}$ repeating themselves downward, where i is the exponent of 2 of the binary notation in the rows. Let us call them 01-*strings*. For example, in the third column, that is, $i = 2$, we see $\underline{00001111}00001\ldots$ The bit string from 0 to $n-1$ has length n and the length of the 01-strings is 2^{i+1}, hence the number of 01-strings it contains is $\lfloor n/2^{i+1} \rfloor$. For example, if $n = 13$, that is, $n = (1101)_2$, and $i = 1$, the number of 01-strings is $\lfloor 13/2^2 \rfloor = 3$. Since half the bits in a 01-string are 1-bits, each of these strings contains 2^i 1-bits, for instance, there are at least $3 \cdot 2^1 = 6$ such bits between 0 and 12, on the second column from the right. We wrote "at least" because there may be other 1-bits which could not make up half a complete 01-string, as, for instance, in the bit string from 0 to 10 included, when $i = 1$. This happens when n is not a power of 2. These extra bits can at most make up a 1-string of length $2^i - 1$, which is the greatest remainder of the Euclidean division by 2^i. Let us define $\sigma_{n-1,i}$ as the sum of the bits of the ith column, where the column 0 is the rightmost, from the row 0 to $n-1$. Then

$$\left\lfloor \frac{n}{2^{i+1}} \right\rfloor 2^i \leqslant \sigma_{n-1,i} \leqslant \left\lfloor \frac{n}{2^{i+1}} \right\rfloor 2^i + (2^i - 1).$$

By means of $x - 1 < \lfloor x \rfloor \leqslant x$, we deduce the weaker, but simpler

$$n/2 - 2^i < \sigma_{n-1,i} \leqslant n/2 + 2^i - 1. \tag{16.69}$$

Remains to sum on all the columns so the leftmost bit of $n-1$ is accounted for. Equation (13.35) on page 301 tells us that n is made of $\lfloor \lg n \rfloor + 1$ bits, so its leftmost bit is at position $\lfloor \lg n \rfloor$. Accordingly, the leftmost bit of $n-1$ is at position $\lfloor \lg(n-1) \rfloor$. This expression can be simplified by a simple variation of the argument that led to equation (13.35). Let us assume that n is made of m bits, that is, $n := (b_{m-1} \ldots b_0)_2$.

$$\begin{gathered} 2^{m-1} \leqslant n \leqslant 2^m - 1 \Rightarrow 2^{m-1} < 2^{m-1} + 1 \leqslant n + 1 \leqslant 2^m \\ \Rightarrow m - 1 < \lg(n+1) \leqslant m \Rightarrow \lceil \lg(n+1) \rceil = m = \lfloor \lg n \rfloor + 1. \end{gathered}$$

Substituting $n-1$ for n leads to $\lfloor \lg(n-1) \rfloor = \lceil \lg n \rceil - 1$. By summing all the sides of inequations (16.69) from $i = 0$ to $\lceil \lg n \rceil - 1$, we cover all the 1-bits of all the binary expansions of the integers in $[0..n-1]$:

$$\sum_{i=0}^{\lceil\lg n\rceil-1}\left(\frac{n}{2}-2^i\right) < \sum_{i=0}^{\lceil\lg n\rceil-1}\sigma_{n-1,i} \leqslant \sum_{i=0}^{\lceil\lg n\rceil-1}\left(\frac{n}{2}+2^i-1\right).$$

The sum in the middle is none other than $\overline{\mathcal{B}}(n) = \sum_{i=0}^{n-1}\nu_i$, therefore

$$\tfrac{1}{2}n\lceil\lg n\rceil - 2^{\lceil\lg n\rceil} + 1 < \overline{\mathcal{B}}(n) \leqslant \tfrac{1}{2}n\lceil\lg n\rceil + 2^{\lceil\lg n\rceil} - \lceil\lg n\rceil - 1.$$

Using the inequalities $x \leqslant \lceil x\rceil < x+1$ leads to

$$\tfrac{1}{2}n\lg n - 2n + 1 < \overline{\mathcal{B}}(n) \leqslant \tfrac{1}{2}n\lg n + \tfrac{5}{2}n - \lg n - 1.$$

The same approach, based upon delay trees, leads to

$$\overline{\mathcal{W}}(2p) = \overline{\mathcal{W}}(2p-1) + \nu_{2p-1};\quad \overline{\mathcal{W}}(2p+1) = \overline{\mathcal{W}}(2p) + 2^{\rho_{2p}} + \nu_{2p} - 1.$$

In Question 2, we derived equation (16.68): $\overline{\mathcal{W}}(2^p) = \overline{\mathcal{W}}(2^p-1)+p$, which is consistent with the recurrent equation for $\overline{\mathcal{W}}(2p)$, but $\overline{\mathcal{W}}(n+1) = \overline{\mathcal{W}}(n) + \nu_n$ does *not* hold for even values of n.

Question 8. Skipped.

Question 9. If $n = 2^p$, then $\lfloor\lg n\rfloor = \lceil\lg n\rceil = \lg n$, thus $n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+1} = n\lg n = n\lceil\lg n\rceil + n - 2^{\lceil\lg n\rceil}$; otherwise, $\lfloor\lg n\rfloor + 1 = \lceil\lg n\rceil$, thus $n\lfloor\lg n\rfloor + 2n - 2^{\lfloor\lg n\rfloor+1} = n\lceil\lg n\rceil + n - 2^{\lceil\lg n\rceil}$. QED. This identity can be used to derive a legible upper bound for $\mathcal{W}(n)$ and $\mathcal{A}(n)$:

$$\begin{aligned}\mathcal{W}(n) &\leqslant n\lfloor\lg n\rfloor + 3n - 2^{\lfloor\lg n\rfloor+1} - 1 = n\lceil\lg n\rceil + 2n - 2^{\lceil\lg n\rceil} - 1\\ &\leqslant n\lceil\lg n\rceil + n - 1.\end{aligned}$$

Question 10. We have $\mathcal{A}_n^{\mathsf{sort}} \geqslant n\lg n - (\alpha-1)n + 2\lg n + 9/2$. A computer algebra system leads to $1 \leqslant n \leqslant 26 \Rightarrow \mathcal{A}_n^{\mathsf{i2wb}} < \mathcal{A}_n^{\mathsf{sort}}$.