Algebra of Programming

Paulo R. Pereira

Preface

One of the main objectives of these notes is to offer a theoretical and practical support to the *Algebra of Programming* course unit [Oliveira], included in the Computer Science and Engineering courses at the University of Minho. Its main reference is the book *Program Design by Calculation* [Oliveira, 2022] of Prof. José N. Oliveira, regent of the mentioned course unit.

University of Minho, Braga, Oct. 2021

Paulo R. Pereira

Contents

Chapter 1

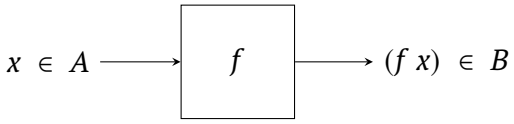
Introduction to pointfree functional programming

1.1 Functions and Types

A function

$$f : A \to B \tag{1.1}$$

is an abstraction of the following process: it is a "black box" that produces a specific result from a given input.



The box itself is not really relevant to describe such a process. What is truly important is the function's name, the type of the input data and the type of the output data. Therefore, the process can be described by a simple arrow

$$A \xrightarrow{f} B$$

This translates to the following "contract":

> f *commits itself* to producing a value of type B as long as it is provided with a value of type A.

This contract corresponds to the *signature* or *type* of the function, expressed by the arrow $A \to B$. It has become standard to use arrows to represent function signatures (or types). Therefore, in order to indicate that the function f accepts arguments of type A and produces

results of type B, the following equivalent notations are used:

$$f : A \to B, \quad A \xrightarrow{f} B\,, \quad f : B \to A \quad \text{ou} \quad B \xleftarrow{f} A$$

This corresponds to writing `f :: a -> b` in the functional programming language Haskell, where type variables are expressed using lowercase letters. A will be referred to as the domain of f, and B as the codomain of f. Both A and B are symbols or expressions that denote sets of values, more commonly known as types.

How are the values at the output of a function produced? To answer this question, we have to inspect the inside of the "black box" to find out what the "rule of computation" of the function is. For example,

$$\begin{aligned} &succ : \mathbb{N} \to \mathbb{N} \\ &succ\ n \stackrel{\text{def}}{=} n + 1 \end{aligned}$$

expresses the rule of "finding the next natural number" — the successor of n is $n + 1$.

The definition of the *rule of computation* is the most frequent approach to defining a function, i.e, that of directly specifying its behaviour at an arbitrary point of the domain, which is called a variable. In the example given above, it is specified directly on the arbitrary point n of the domain of $succ$ ($\mathbb{N}$) that the behaviour of $succ$ at that point is to add one unit to it. This approach is called the *pointwise* definition — one defines a "rule" that is applied to all points in the domain.

On the other hand, *pointfree* definitions are characterised by the absence of points (or variables). Functions are defined via a combination of other, simpler functions, using a limited set of combinators. The choice of these combinators is dictated by the power of the calculus laws that are associated with them. One of the main objectives of this approach is to facilitate compositional programming, one of the bases of the construction of software. The composition of functions will be the first combinator to to be studied in the section 1.5.

1.2 Function application

The purpose of functions is to be applied to arguments in order to obtain results. This application is expressed by juxtaposition. For example, for $f : A \to B$ and $a \in A$, the application of f to the argument a is denoted as $f\ a$.

It is also important to note that function application associates to the left. That is, for $g : A \to B \to C$, where $a \in A$ and $b \in B$, $g\ a\ b$ represents $(g\ a)\ b$ and not $g\ (a\ b)$. For this reason, $g : A \to B \to C$ is an abbreviation of $g : A \to (B \to C)$ and not $g : (A \to B) \to C$.

Functions like $g : A \to B \to C$ are often referred to within the context of functional programming as functions that receive "multiple arguments sequentially". Such functions are called

higher-order functions and will be discussed in more detail later on. However, it is essential to introduce some concepts in this context.

1.3 Introduction to higher-order functions

Let us consider the function `take :: Int -> [a] -> [a]`, which returns the largest prefix of a given list whose size does not exceed n, as an example.

```
> take 3 []
[]
> take 3 [1,2,3,4,5]
[1,2,3]
> take 7 [1,2,3,4,5]
[1,2,3,4,5]
```

As mentioned earlier, `take :: Int -> [a] -> [a]` is a simplified form of the signature `take :: Int -> ([a] -> [a])`. In other words, the function first takes an integer `n`, and then, if the process allows, the function `take n :: [a] -> [a]` is executed, resulting in the mentioned specification. Therefore, we may think of a family of functions $take_n : [a] \rightarrow [a]$ indexed by the integer n, given by $take_n = take\ n$.

For example,

```
> f = take 3
:t f
f :: [a] -> [a]
> f [1,2,3,4,5]
[1,2,3]
>
> g = take 7
:t g
g :: [a] -> [a]
> g [1..10]
[1,2,3,4,5,6,7]
```

Consider also the function `filter :: (a -> Bool) -> [a] -> [a]` which returns the largest sublist whose elements satisfy the predicate `p :: (a -> Bool)`. There is also a family of functions $filter_p :: [a] \rightarrow [a]$ indexed by the predicate p given by $filter_p = filter\ p$. For example,

```
> :t even
even :: Integral a => a -> Bool
> even 7
False
```

```
> even 10
True
>
> getEven = filter even
> :t getEven
getEven :: Integral a => [a] -> [a]
> getEven [1..10]
[2,4,6,8,10]
```

TBC

1.4 Functional equality

Two functions $f, g : A \to B$ are equal if they "agree" at the pointwise-level, that is

$$f = g \quad \textit{iff} \quad \langle \forall a : a \in A : f\ a =_B g\ a \rangle \tag{1.2}$$

where $\langle \forall x : R : T \rangle$ means "for all x in the range R, the term T holds", where R and T are logical expressions involving x, and $=_B$ denotes equality in the set B.

1.5 Functional composition

This is the first functional combinator to be studied in this course. Let f and g be the following functions:

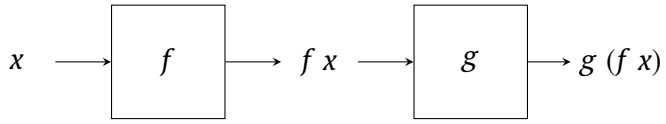
$$A \xrightarrow{\ f\ } B$$

$$B \xrightarrow{\ g\ } C$$

"Chaining" the two arrows, one obtains another function $g \cdot f : A \to C$

$$A \xrightarrow{\ f\ } B \xrightarrow{\ g\ } C \qquad A \xrightarrow[g.f]{} C$$

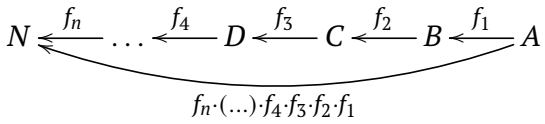
called the *composition* of g and f.



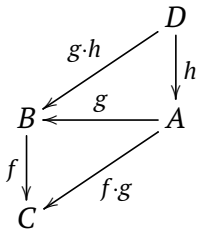
One reads it as g *after* f and it is defined in the usual way:

$$(g \cdot f)\ x = g\ (f\ x) \tag{1.3}$$

Functional composition is one of the foundations of this discipline. We can functionally compose n functions, provided that the output type of each function matches the input type of the "next function".



Being the most basic of all functional combinators, most programmers don't even think of it when they want to combine or chain functions to get a more elaborate program. This is due to one of its most important properties, the *associative* property of composition:



$$(f \cdot g) \cdot h = f \cdot (g \cdot h) \tag{1.4}$$

Another important property is the law of *Leibniz*,

$$f \cdot h = g \cdot h \Leftarrow f = g \tag{1.5}$$

which states that, given the equality of two functions, the pre-compositions of them with another function also form an equality — the opposite is not true.

Problem 1 A mobile phone manufacturer has submitted the following requirements for the storage operation of a phone call:

(a) the most recent calls appear first;

(b) no repeated calls appear;

(c) only the last 10 entries are stored.

Write a definition of the operation `store :: Call -> [Call] -> [Call]` which adds a call to a list of calls according to the given criteria, using only the variable of type `Call`.

Solution

```
store c = take 10 . (c :) . filter (/= c)
```

1.6 Identity functions

Identity functions are those that simply copy the input to the output: $f\ a = a$, for $f : A \to A$ and $a \in A$. In this case, f is called the identity function on A:

$$\begin{aligned} &id_A : A \to A \\ &id_A\ a \stackrel{\text{def}}{=} a \end{aligned} \tag{1.6}$$

As expected, every type X has its identity function id_X. However, subscripts will be omitted whenever they are implicit in the context. We can thus assume a "single" identity function.

At first glance, the identity function may not seem very useful or interesting. However, it becomes important when it comes to preserve information and also in its interaction with functional composition, as captured by its natural property:

$$f \cdot id = id \cdot f = f \tag{1.7}$$

1.7 Constant functions

Unlike the identity function, which does not lose any information, constant functions lose all (or almost all) information. Regardless of the input data, the output is always the same value. For example, if g is the constant function that produces the number 23, then:

```
> g 7
23
> g "hello there!"
23
> g 23
23
> g []
23
> g 'p'
23
> g [1..]
23
```

Thus, let C be a non-empty datatype and $c \in C$. The constant function *everywhere c*, for an arbitrary type A, is defined as follows:

$$\begin{aligned} &\underline{c} : A \to C \\ &\underline{c}\ a \stackrel{\text{def}}{=} c \end{aligned} \tag{1.8}$$

Therefore, the example function g is given by $g = \underline{23}$. In Haskell, constant functions are defined using the function `const :: a -> b -> a` from the Standard Prelude. Thus, $g = \underline{c}$ is defined in Haskell as `g = const c`.

Finally, the following properties are quite intuitive,

$$\begin{array}{ccc} C & \xleftarrow{\underline{c}} & A \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle f} \\ C & \xleftarrow[\underline{c}]{} & B \end{array} \qquad \underline{c} \cdot f = \underline{c} \tag{1.9}$$

$$f \cdot \underline{c} = \underline{f\ c} \tag{1.10}$$

known respectively as the *natural-constant* property and the *constant-fusion* property.

Note that in the diagram of property 1.9, the symbol $\underline{c}$ denotes two different functions: $\underline{c_A} : A \to C$ and $\underline{c_B} : B \to C$. Thus, just like with identity functions, subscripts will be omitted whenever they are implicit in the context.

As we have seen, identity functions and constant functions are the limits in the "functional spectrum" regarding data preservation. The identity function preserves all the values it receives, while constant functions lose all (or almost all) the data. Any other function lies "in between" these two functions, losing some of the input data. How does a function lose information? Essentially, in two ways: by "confusing" arguments and mapping them to the same output, or by simply "ignoring" values from its domain. Such functions will be the subject of study in the next sections.

1.8 Monomorphisms

A monomorphism (or injective function) is a function that "does not confuse arguments", i.e., it does not produce the same result for two different inputs.

For example, let g be the function that calculates the square of an integer. Given that $x = 4$, what is the value of x? It could be 2, but it could also be -2. In other words, g confuses the two arguments 2 and -2, as even though they are different values, the function maps both values to the same result.

Therefore, a function $B \xleftarrow{f} A$ is said to be a *monomorphism* if, for any pair of functions $A \xleftarrow{h,k} C$, if $f \cdot h = f \cdot k$ then $h = k$, cf.

$$B \xleftarrow{f} A \underset{k}{\overset{h}{\leftleftarrows}} C$$

We say that f is "post-cancellable".

1.9 Epimorphisms

An epimorphism (or surjective function) is a function that does not "ignore values" from its codomain. In other words, if $f : A \to B$ is an epimorphism, for any value $b \in B$, there is at least one $a \in A$ such that $b = f(a)$.

It is easy to verify that most constant functions are not epimorphisms, except for those whose codomain is a singleton set, i.e., a set with only one element. Consider, for example, the function $g = \underline{23}$ with the codomain $\mathbb{N}_0$. The function only produces the value 23, and all other natural numbers are ignored.

Therefore, a function $A \xleftarrow{f} B$ is said to be an *epimorphism* if, for any pair of functions $C \xleftarrow{h,k} A$, if $h \cdot f = k \cdot f$ then $h = k$, cf.

$$C \underset{k}{\overset{h}{\leftleftarrows}} A \xleftarrow{f} B$$

We say that f is "pre-cancellable".

1.10 Isomorphisms

An isomorphism is a function that is both a monomorphism and an epimorphism. The term isomorphism derives from the Greek words "isos" meaning "equal" and "morphe" meaning "form". This concept conveys the idea of a function that maps values from one type A to an "equivalent" type B, i.e., a type with the "same form". Thus, a function $f : A \to B$ that is an isomorphism always has an inverse function $f^\circ : B \to A$ such that

$$f \cdot f^\circ = id_B \ \wedge \ f^\circ \cdot f = id_A, \tag{1.11}$$

which means that f is invertible.

Isomorphisms are highly significant as they can convert data from one particular "format" (e.g., A) to another "format" (e.g., B) without losing information. These formats contain the same information but differently organized. We say that A is isomorphic to B and write $A \cong B$ to express this fact. Isomorphic data domains are considered abstractly the same. For example, let us recall the distributive property of elementary algebra:

$$x\ (y + z) = x\ y + x\ z$$

A number written in the format $x\ (y + z)$ can be rewritten in the format $x\ y + x\ z$, and vice versa, without any alteration. The expressions $x\ (y+z)$ and $x\ y+x\ z$ are equivalent "formats" in numerical representation.

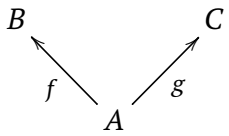
Isomorphisms also provide flexibility in pointfree calculus. If, for some reason, f° is easier to algebraically manipulate than f, then the following rules, known as "shunting rules",

$$\begin{aligned} f \cdot g = h &\equiv g = f^\circ \cdot h \\ g \cdot f = h &\equiv g = h \cdot f^\circ \end{aligned} \tag{1.12}$$

will be of great help.

1.11 Product of functions

In section 1.5, functional composition was presented as one of the basis for combining functions to construct more complex ones. However, not all functions can be combined using composition. For example, functions $f : A \to B$ and $g : A \to C$ are a case where functional composition cannot be applied because the input type of one function does not match the output type of the other. However, since both functions share the same input type, they can be combined in the following way:



resulting in the set of pairs $(f\ a, g\ a)$ for each $a \in A$. This set is given by the cartesian product of B with C, i.e., the set

$$B \times C = \{(b, c) \mid b \in B\ \wedge\ c \in C\}$$

This operation of pairing outputs from functions f and g that share the same domain will be the new combinator $\langle f, g\rangle$ defined as follows:

$$\begin{aligned} &\langle f, g\rangle : A \to B \times C \\ &\langle f, g\rangle\ a \stackrel{\text{def}}{=} (f\ a, g\ a) \end{aligned} \tag{1.13}$$

and pronounced "f *split* g".

The $\langle f, g\rangle$ combinator captures the information of both functions f and g in a similar way to how the cartesian product $A \times B$ captures the types A and B. This means that, just like the types A and B can be extracted from the cartesian product using the projections π_1 and π_2,

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

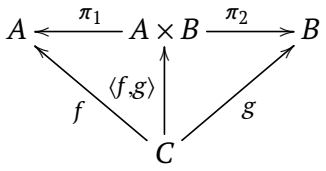
whose definitions are

$$\pi_1\ (a, b) = a\ \ \wedge\ \ \pi_2\ (a, b) = b \tag{1.14}$$

functions f and g can also be extracted from $\langle f, g\rangle$ using the same projections:

$$\pi_1 \cdot \langle f, g\rangle = f\ \ \wedge\ \ \pi_2 \cdot \langle f, g\rangle = g \tag{1.15}$$

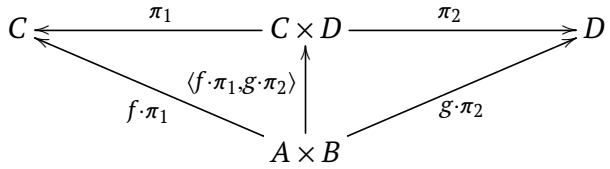
This fact corresponds to the $\times$**-cancellation** property expressed in the following diagram:



What if functions f and g do not share the same domain?

$$\begin{array}{l} C \xleftarrow{\ f\ } A \\ D \xleftarrow{\ g\ } B \\ \hline \ldots ? \end{array}$$

The $\langle f, g\rangle$ combinator cannot be used in this case. However, the projections π_1 and π_2 over the product $A \times B$ still allow us to use the *split* combinator as follows:



The resulting function $\langle f \cdot \pi_1, g \cdot \pi_2\rangle$ is mapping $A \times B$ to $C \times D$, which corresponds to the parallel application of f with g and will be expressed as $f \times g$. Thus, by definition,

$$f \times g \stackrel{\text{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2\rangle \tag{1.16}$$

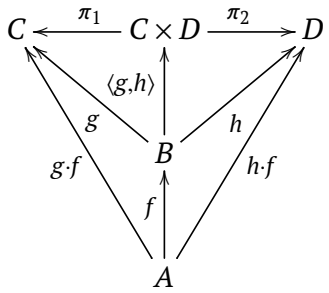
$$\equiv \quad \{\ \text{extensional equality (1.2)}\ \}$$

$$(f \times g)\ (a, b) \stackrel{\text{def}}{=} (f\ a, g\ b) \tag{1.17}$$

and is pronounced"product of f with g".

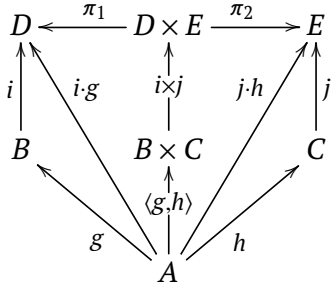
What results from the interaction between the composition $f \cdot g$, the split $\langle f, g\rangle$ and the product $f \times g$ combinators? Several properties, which, together with the aforementioned $\times$**-cancellation** (1.15), form the laws of functional calculus with respect to the product:

- $\times$**-fusion**, which relates composition with *split*:



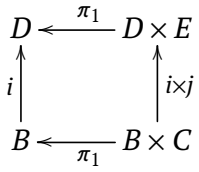
$$\langle g, h\rangle \cdot f = \langle g \cdot f, h \cdot f\rangle \tag{1.18}$$

- $\times$**-absorption**, in which "*split* absorbs $\times$":



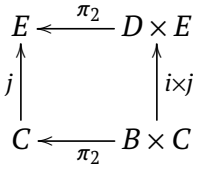
$$(i \times j) \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle \tag{1.19}$$

- **natural-π_1**:



$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{1.20}$$

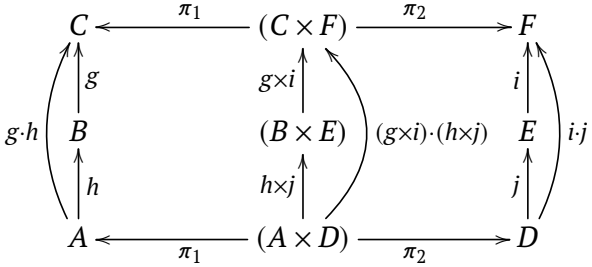
- **natural-π_2**:



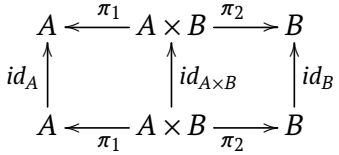
$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{1.21}$$

- $\times$**-functor**, which expresses a "bi-distribution" of $\times$ through composition:

$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \tag{1.22}$$

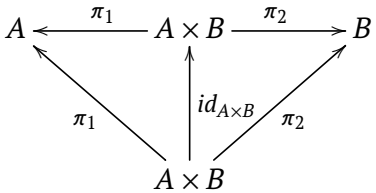


- $\times$**-id-functor**:



$$id_A \times id_B = id_{A \times B} \tag{1.23}$$

- $\times$**-reflection**, where the *split* combinator is constructed solely using the projections π_1 and π_2:



$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \tag{1.24}$$

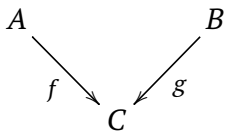
```
-- projections
p1 = fst
p2 = snd

split          :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g a    = (f a, g a)

infix 5  ><
(><)       :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
(><) f g   = split (f . p1) (g . p2)
```

Problem 2 Present a *pointfree* definition of the function `average :: [Int] -> Int` which calculates the average of a list of integers.

Solution `average = uncurry div . split sum length`

1.12 Coproduct of functions

As we saw in the previous section, the functional combinator *split* was created to combine functions that do not meet the requirements of function composition but share the same domain. The "dual" situation corresponds to functions that share the same codomain, that is, they have the same output type.



Both functions f and g produce values of type C. Therefore, it is intuitive to think of a combinator that executes f or g depending on "which side we are on" — either we are on "side A" and execute f, or we are on "side B" and execute g. Let us call this combinator $[f, g]$. The codomain of $[f, g]$ will be C. Now, what should be its domain? Well, it should be a datatype that represents "either A or B". For this, we might consider using $A \cup B$. However, when the intersection $A \cap B$ is non-empty, it is not possible to determine if a given element $x \in A \cap B$ came from "side A" or "side B". Therefore, we need to use the concept of *disjoint union*,

$$A + B \stackrel{\text{def}}{=} \{i_1\ a \mid a \in A\} \cup \{i_2\ b \mid b \in B\}$$

assuming the "tagging" functions i_1 and i_2, whose signatures are $A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$, which associate a different tag to ensure that values of type A and values of type B do not mix in the set $A + B$. Functions i_1 and i_2 are called the injections of the disjoint union, and it is said that i_1 "injects" values on the left, while i_2 "injects" values on the right.

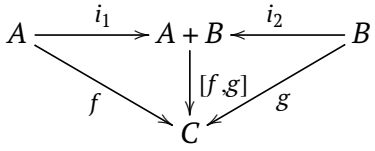
In the Haskell Standard Prelude, the datatype $A + B$ is given by

```
data Either b c = Left b | Right c
```

which means that `Left` and `Right` correspond to the injections i_1 and i_2 respectively. So, based on the disjoint union $A + B$, also known as the *coproduct* of A and B, the combinator $[f, g]$ is defined, pronounced as "*either* f *or* g":

$$\begin{aligned} &[f,g] : A + B \to C \\ &[f,g]\ x \stackrel{\text{def}}{=} \begin{cases} x = i_1\ a \Rightarrow f\ a \\ x = i_2\ b \Rightarrow g\ b \end{cases} \end{aligned} \tag{1.25}$$

Just as we did with the product, we can express this combinator in a diagram:

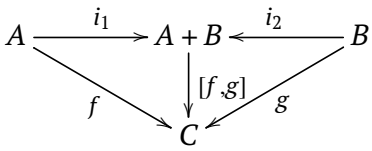


It is interesting to note how similar the diagrams of the product and the coproduct are — we simply invert the arrows, replace the projections with injections and the *split* with the *either*. This reflects the fact that the product and the coproduct are dual constructions in mathematics (just like sine and cosine in trigonometry). This duality leads to a significant economy, as everything said about the product $A \times B$ can be dualized to the coproduct $A + B$. For example, the sum of functions $f + g$ is given by the dual notion of the product $f \times g$:

$$f + g \stackrel{\text{def}}{=} [i_1 \cdot f, i_2 \cdot g] \tag{1.26}$$

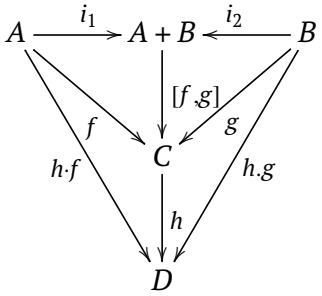
The following properties also contribute as evidence of the mentioned duality:

- **+-cancellation**:



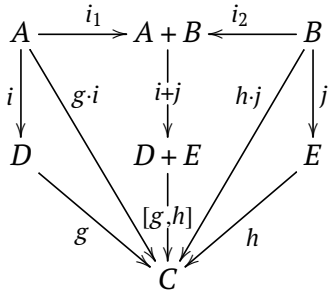
$$\begin{cases} [f,g] \cdot i_1 = f \\ [f,g] \cdot i_2 = g \end{cases} \tag{1.27}$$

- **+-fusion**:



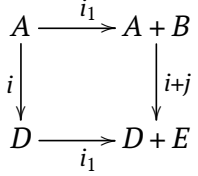
$$h \cdot [f,g] = [h \cdot f, h \cdot g] \tag{1.28}$$

- +**-absorption**:



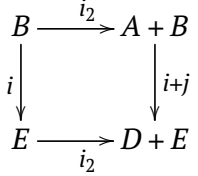
$$[g,h]\cdot(i+j) = [g\cdot i, h\cdot j] \tag{1.29}$$

- **natural-**i_1:



$$(i+j)\cdot i_1 = i_1 \cdot i \tag{1.30}$$

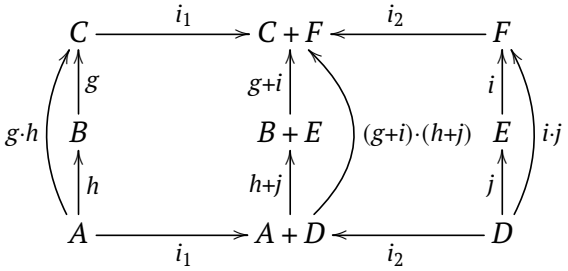
- **natural-**i_2:



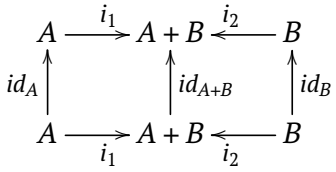
$$(i+j)\cdot i_2 = i_2 \cdot j \tag{1.31}$$

- +**-functor**:

$$(g\cdot h) + (i\cdot j) = (g+i)\cdot(h+j) \tag{1.32}$$

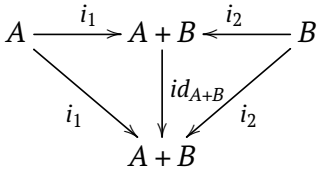


- +**-id-functor**:



$$id_A + id_B = id_{A+B} \tag{1.33}$$

- +**-reflection**:



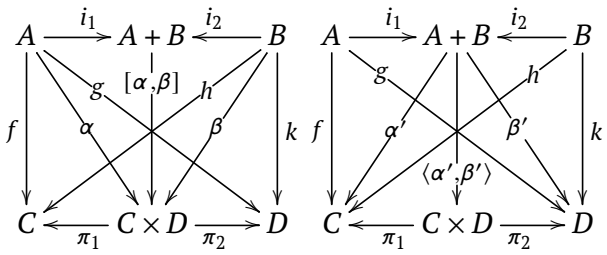
$$[i_1, i_2] = id_{A+B} \tag{1.34}$$

```haskell
-- injections
i1 = Left
i2 = Right

-- either is defined in 'Data.Either'
either                  :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left a)     =  f a
either _ g (Right b)    =  g b

infix 4  -|-
(-|-)       :: (a -> b) -> (c -> d) -> Either a c -> Either b d
(-|-) f g   = either (i1 . f) (i2 . g)
```

1.13 The exchange law

A function that maps values from a coproduct $A + B$ to values of a product $A' \times B'$ can be alternatively expressed as an either or a split. Both expressions are equivalent, and this equivalence is captured by the *exchange law*:

$$[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle \tag{1.35}$$



$$\sigma = [\alpha, \beta] \qquad \sigma' = \langle \alpha', \beta' \rangle$$
$$\alpha = \langle f, g \rangle \qquad \alpha' = [f, h] \qquad \sigma = \sigma'$$
$$\beta = \langle h, k \rangle \qquad \beta' = [g, k]$$

1.14 Natural properties

$$\begin{array}{ccc} A & \mathsf{F}\,A \xleftarrow{\phi} \mathsf{G}\,A \\ {\scriptstyle f}\downarrow & {\scriptstyle \mathsf{F}\,f}\downarrow \qquad\qquad \downarrow{\scriptstyle \mathsf{G}\,f} \\ B & \mathsf{F}\,B \xleftarrow[\phi]{} \mathsf{G}\,B \end{array} \qquad (\mathsf{F}\,f)\cdot\phi = \phi\cdot(\mathsf{G}\,f) \tag{1.36}$$

1.15 Universal properties

TBC

1.16 McCarthy's conditional

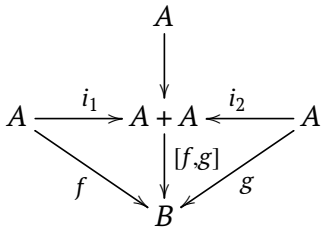
Most functional programming languages provide pointwise conditional expressions in the form

$$\textbf{if } p\ x \textbf{ then } f\ x \textbf{ else } g\ x \tag{1.37}$$

which translates into the following process: given a predicate $p : A \to \mathbb{B}$ and two functions $f, g : A \to B$, if $p\ x$ holds then the process results in $f\ x$, otherwise in $g\ x$, i.e.,

$$\begin{cases} p\ x \Rightarrow f\ x \\ \neg\ (p\ x) \Rightarrow g\ x \end{cases}$$

Can we rewrite the expression (1.37) in a pointfree style? Let us start by assuming that $p\ x$ has already been calculated. In this case, either f is executed or g is executed. Now, this latter operation is clearly the combinator $[f, g]$.



This means that if $p\ x$ holds, the argument is injected on the left side, so that the operation $f\ x$ is executed. Otherwise, the argument is injected on the right side, and $g\ x$ is executed

instead of $f\ x$. Therefore, let us denote this operation as $(p?)$. It is quite clear that the expression (1.37), rewritten in a pointfree style, is given by

$$[f, g] \cdot p? \tag{1.38}$$

What can be said about $(p?)$? Let us start by introducing the following isomorphism:

$$\mathbb{B} \times A \underset{\alpha}{\overset{\alpha^\circ}{\cong}} A + A \qquad \alpha = [\langle \underline{True}, id \rangle, \langle \underline{False}, id \rangle] \tag{1.39}$$

whose function α° will be of great utility in the definition of $p?$, as it precisely produces what $[f, g]$ takes as input. Therefore, let us calculate α° from $\alpha^\circ \cdot \alpha = id$

$$\alpha^\circ \cdot \alpha = id \tag{1.40}$$

$\equiv$ { definition of α }

$$\alpha^\circ \cdot [\langle \underline{True}, id \rangle, \langle \underline{False}, id \rangle] = id \tag{1.41}$$

$\equiv$ { +-fusion (1.28); +-reflection (1.34); +-eq (**??**) }

$$\begin{cases} \alpha^\circ \cdot \langle \underline{True}, id \rangle = i_1 \\ \alpha^\circ \cdot \langle \underline{False}, id \rangle = i_2 \end{cases} \tag{1.42}$$

$\equiv$ { extensional equality (1.2); composition (1.3); constant-def (1.8); natural-id (1.7) }

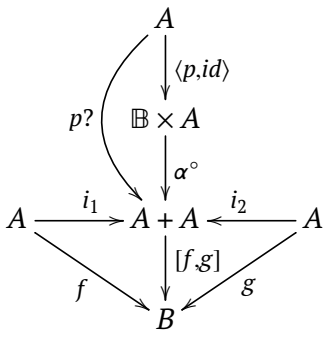
$$\begin{cases} \alpha^\circ\ (True, x) = i_1\ x \\ \alpha^\circ\ (False, x) = i_2\ x \end{cases} \tag{1.43}$$

The goal now is for the function α° to receive the pair $(p\ x, x)$, in order to inject the argument into the corresponding "side" dictated by the result of $p\ x$. To do this, we simply execute $\langle p, id \rangle$, as follows:

$$A \xrightarrow{\langle p, id \rangle} \mathbb{B} \times A \xrightarrow{\alpha^\circ} A + A \qquad A \xrightarrow{p?} A + A$$

The function $p?$ is referred to as *the guard* associated with the predicate p. In a way, the guard $p?$ is much more informative than the predicate itself, as it already gives us the result of testing p on a given input.

Finally, the composition (1.38) can be represented by the following complete diagram:



corresponds to the well-known functional combinator "McCarthy's conditional" and is usually denoted by the expression $p \to f, g$. Therefore,

$$p \to f, g \stackrel{\text{def}}{=} [f, g] \cdot p? \tag{1.44}$$

The use of the *either* combinator suggests that, when reasoning about conditionals, one can turn to the algebra of coproducts as a potential aid. In fact, the first fusion law for conditionals

$$h \cdot (p \to f, g) = p \to h \cdot f, h \cdot g \tag{1.45}$$

is a direct consequence of the +-fusion law (1.28). The second fusion law also results from another law of coproducts, namely the +-absorption (1.29).

$$(p \to f, g) \cdot h = (p \cdot h) \to f \cdot h, g \cdot h \tag{1.46}$$

Note that before applying the absorption law, it was necessary to apply the natural property of the guard:

$$p? \cdot f = (f + f) \cdot (p \cdot f)? \tag{1.47}$$

```
grd      :: (a -> Bool) -> a -> Either a a
grd p x = if p x then i1 x else i2 x

-- McCarthy's conditional:
cond       :: (b -> Bool) -> (b -> c) -> (b -> c) -> b -> c
cond p f g = (either f g) . (grd p)
```

Problem 3 Present a *pointfree* definition of the function out_{List}, given by

```
outList []     = i1 ()
outList (h:t) = i2 (h, t)
```

Solution `outList = (const () -|- split head tail) . grd (== [])`

1.17 Exponentials

Given a function $f : C \times A \to B$, we want to construct a family of functions of type $A \to B$ according to the following approach: for each $c \in C$, we construct the function

$$\begin{aligned} &f_c : A \to B \\ &f_c\ a \stackrel{\text{def}}{=} f\ (c, a) \end{aligned} \tag{1.48}$$

In other words, the construction of this family is a function of type $C \to (A \to B)$, meaning that given a $c \in C$, it produces a function of type $A \to B$. Functions of this type are called higher-order functions — functions that not only produce functions but also receive functions as arguments. To represent the type $A \to B$ (or $B \leftarrow A$), we shall use the notation B^A. Thus,

$$B^A \stackrel{\text{def}}{=} \{g \mid g : A \to B\} \tag{1.49}$$

corresponds to the set inhabited by functions from A to B, which means that the functional declaration $g : A \to B$ is equivalent to $g \in B^A$. And since the purpose of functions is to be applied to arguments, the introduction of the *apply* combinator is quite intuitive:

$$\begin{aligned} &ap : B^A \times A \to B \\ &ap\ (f, a) \stackrel{\text{def}}{=} f\ a \end{aligned} \tag{1.50}$$

Now, going back to the function $f : C \times A \to B$, let us recall the strategy of, for each $c \in C$, producing a function $f_c \in B^A$. As mentioned, this process corresponds to a function of type $C \to B^A$, which expresses f as a family of functions of type $A \to B$ indexed by the type C. Such functions will be referred to as transposes, and the notation $\overline{f}$ will be used to represent them, read as "transpose of f". As expected, f and $\overline{f}$ are mutually related by the following property:

$$f\ (c, a) = (\overline{f}\ c)\ a \tag{1.51}$$

The rule is

$$\frac{B \xleftarrow{f} C \times A}{B^A \xleftarrow{\overline{f}} C}$$

However, despite the equality, $\overline{f}$ has the advantage of being more "tolerant" than f. While f requires both arguments in the pair (c, a), $\overline{f}$ "settles" for receiving the argument c first, and later, if the process allows, the argument a.

Just like the product $A \times B$ and the coproduct $A + B$, the exponentiation B^A also has a universal property,

$$k = \overline{f} \quad \Leftrightarrow \quad f = ap \cdot (k \times id) \tag{1.52}$$

from which the following laws can be derived:

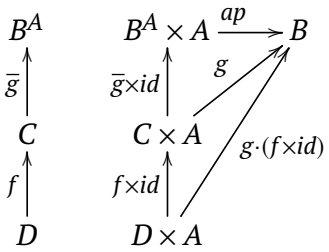
- **Exp-cancellation**:

$$\begin{array}{ccc} B^A & B^A \times A \xrightarrow{ap} B & \\ \overline{f}\uparrow & \overline{f}\times id\uparrow \quad \nearrow f & \\ C & C \times A & \end{array} \qquad f = ap \cdot (\overline{f} \times id) \tag{1.53}$$

- **Exp-reflection**:

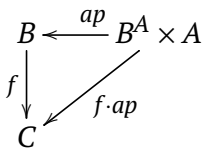
$$\begin{array}{ccc} B^A & B^A \times A \xrightarrow{ap} B & \\ id_{B^A}\uparrow & id_{B^A}\times id_A\uparrow \quad \nearrow ap & \\ B^A & B^A \times A & \end{array} \qquad \overline{ap} = id_{B^A} \tag{1.54}$$

- **Exp-fusion**:



$$\overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{1.55}$$

The remaining laws of exponentiation, such as the absorption law, require the new functional combinator that arises from the function $\overline{f \cdot ap}$. What will be the signature of this function? The diagram of $f \cdot ap$:



leaves no doubt that the answer is $\overline{f \cdot ap} : B^A \to C^A$. The notation to be used to express the combinator $\overline{f \cdot ap}$ will be f^A, which follows the rule

$$\frac{C \xleftarrow{f} B}{C^A \xleftarrow{f^A} B^A}$$

But what does this new combinator mean? Well, f^A takes a function $g : A \to B$ as an argument and returns a function of type $A \to C$. Therefore, given a specific $a \in A$, $(f^A\ g)\ a$ will produce a value of type C. It is known that the function f produces values of type C as long as it is given a value of type B. Now, g produces values of type B. Thus, we execute $g\ a$, obtaining a $b \in B$, which is then passed to the function f to produce a $c \in C$. So, what is happening is precisely the functional composition of f with g, i.e., f^A translates to the "composition with f" combinator:

$$f^A\ g \overset{\text{def}}{=} f \cdot g \tag{1.56}$$

i.e.,

$$f^A \overset{\text{def}}{=} (f \cdot) \tag{1.57}$$

In fact,

$$\begin{aligned}
& \overline{f \cdot ap} = f^A \\
= & \quad \{ \text{ exp-universal (1.52) } \} \\
& ap \cdot (f^A \times id) = f \cdot ap \\
= & \quad \{ \text{ extensional equality (1.2); functional composition (1.3) } \} \\
& ap\ ((f^A \times id)\ (g, a)) = f\ (ap\ (g, a))
\end{aligned}$$

$$
\begin{array}{ll}
= & \{\ \times\text{-def (}\textit{pointwise}\text{) (1.17); }\textit{apply}\text{ operator (1.50)}\ \} \\
& (f^A\ g)\ a = f\ (g\ a) \\
= & \{\ \text{functional composition (1.3)}\ \} \\
& (f^A\ g)\ a = (f \cdot g)\ a \\
= & \{\ \text{extensional equality (1.2)}\ \} \\
& f^A\ g = f \cdot g
\end{array}
$$

□

We are now ready to understand the following properties:

- **Exp-absorption**:

$$
\begin{array}{ccc}
D^A & D^A \times A & \xrightarrow{ap} D \\
{\scriptstyle f^A}\uparrow & {\scriptstyle f^A \times id}\uparrow & \uparrow{\scriptstyle f} \\
B^A & B^A \times A & \xrightarrow{ap} B \\
{\scriptstyle \overline{g}}\uparrow & {\scriptstyle \overline{g} \times id}\uparrow & \nearrow{\scriptstyle g} \\
C & C \times A &
\end{array}
\qquad f^A \cdot \overline{g} = \overline{f \cdot g} \tag{1.58}
$$

  Note how the diagram also provides a proof for $f^A = \overline{f \cdot ap}$.

- **Exp-functor**:

$$(g \cdot h)^A = G^A \cdot h^A \tag{1.59}$$

- **Exp-id-functor**:

$$id^A = id \tag{1.60}$$

Let us now return to property (1.51). The chosen notation allows us to express the equality $f\ (a, b) = (\overline{f}\ a)\ b$ through the isomorphism

$$C \times A \to B\ \cong\ C \to B^A$$

which, in turn, can be rewritten as

$$B^{C \times A}\ \cong\ (B^A)^C \tag{1.61}$$

This congruence reminds us of a well-known equality in numerical exponentiation: $b^{c \times a} = (b^a)^c$. Other properties of numerical exponentiation, such as $b^{c+a} = b^c \times b^a$ and $(c \times a)^b = c^a \times c^b$, also have their "corresponding" isomorphisms in functional exponentiation. The first one corresponds to the isomorphism

$$B^{C+A}\ \cong\ B^C \times B^A \tag{1.62}$$

i.e., each pair of functions $(f, g) \in B^C \times B^A$ leads to a unique function $[f, g] \in B^{C+A}$ and vice versa, each function $[f, g] \in B^{C+A}$ leads to two functions $f \in B^C$ and $g \in B^A$. The second property corresponds to the isomorphism

$$(C \times A)^B \cong C^A \times C^B \tag{1.63}$$

which is justified analogously, now using the uniqueness of the $\langle f, g \rangle$ combinator. Such similarities justify the adoption of exponential notation.

Isomorphism (1.61) is at the core of functional programming — in Haskell, we find the predefined functions that compose it:

$$B^{C \times A} \underset{uncurry}{\overset{curry}{\cong}} (B^A)^C \tag{1.64}$$

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f a b = f (a, b)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b
```

This means that *curry* corresponds to transposition in Haskell, that is, $\overline{f}$ corresponds to writing `curry f`, cf. the equivalence

$$\begin{aligned}
& \overbrace{(curry\ f\ a)}^{\overline{f}}\ b = f\ (a, b) \\
\equiv & \quad \{\ \textit{apply}\ \text{operator (1.50)};\ \textit{id}\ \text{definition (1.6)}\ \} \\
& ap\ (\overline{f}\ a, id\ b) = f\ (a, b) \\
\equiv & \quad \{\ \text{product (1.17)};\ \text{composition (1.3)}\ \} \\
& (ap \cdot (\overline{f} \times id))\ (a, b) = f\ (a, b) \\
\equiv & \quad \{\ \text{extensional equality (1.2)}\ \} \\
& ap \cdot (\overline{f} \times id) = f
\end{aligned}$$

from which it follows that the definition of *curry* is a reformulation of the cancellation law (1.53), and therefore,

$$curry\ f \overset{\text{def}}{=} \overline{f} \tag{1.65}$$

Finally, in order to simplify algebraic notation, the inverse of transposition will also have its own notation: *uncurry* f will be abbreviated as $\widehat{f}$, i.e.,

$$g = \overline{f} \Leftrightarrow \widehat{g} = f \tag{1.66}$$

Chapter 2

Introduction to pointfree recursion

2.1 Motivation

TBC

2.2 Inductive datatypes

A datatype is said to be inductive or recursive when it refers to itself in its definition, i.e., T ≅ ...T... (which is written in Haskell as `data T = ... T ...`). A well-known example of this is the case of lists, or finite sequences, already defined in the Haskell Standard Prelude as follows:

```
data [a] = [] | a : [a]
```

A list whose elements are of type A is either empty or constructed by concatenating an element $a \in A$ (usually referred to as the head) with another list (usually referred to as the tail). For example,

```
> []
[]
> 1 : []
[1]
> 1 : 2 : 3 : []
[1,2,3]
```

Let us begin by assuming that this datatype is not yet defined and, therefore, we want to define finite sequences as `data Seq a = ...`. Knowing that a sequence can be either empty

or the construction of a pair head and tail, we shall use the constructors `Nil` and `Cons`:

```
data Seq a = Nil | Cons (a, Seq a)
```

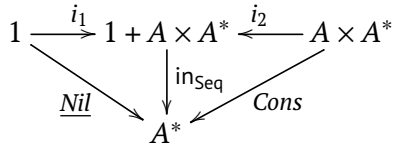
In order for values to be displayed in the usual format, i.e., `[]` for the empty list and `[a,b,c,...]` for `Cons (a, Cons (b, Cons (c, ...)))`, it is necessary to implement an instance of the `Show` class. For example,

```
instance Show a => Show (Seq a) where
  show l = "[" ++ showSeq l ++ "]"
    where
      showSeq Nil = ""
      showSeq (Cons (h, Nil)) = show h
      showSeq (Cons (h, t)) = show h ++ "," ++ showSeq t
```

Thus, using the same examples,

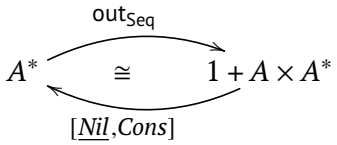
```
> Nil
[]
> Cons (1, Nil)
[1]
> Cons (1, Cons (2, Cons (3, Nil)))
[1,2,3]
```

We shall adopt the notation A^* to represent lists, or finite sequences, of values of type A. This leads to the following diagram:



$$\mathsf{in}_{\mathsf{Seq}} = [\underline{Nil}, Cons]$$

that is,



> **Note**
> Despite the signature of $\underline{Nil}$ being $B \to A^*$, the domain is limited to the singleton type 1 due to the signature of $outSet$. This function is derived from the equation $\mathsf{out}_{\mathsf{Seq}} \cdot \mathsf{in}_{\mathsf{Seq}} = id$,
>
> $$\begin{aligned} & \mathsf{out}_{\mathsf{Seq}} \cdot \mathsf{in}_{\mathsf{Seq}} = id \\ \equiv & \quad \{\ \text{+-fusion (1.28); +-reflection (1.34)}\ \} \end{aligned}$$

$$
\begin{array}{ll}
& [\mathsf{out}_{\mathsf{Seq}} \cdot \underline{Nil}, \mathsf{out}_{\mathsf{Seq}} \cdot Cons] = [i_1, i_2] \\
\equiv & \quad \{ \text{ +-eq (??); extensional equality (1.2); constant (1.8) } \} \\
& \begin{cases} \mathsf{out}_{\mathsf{Seq}}\ Nil = i_1\ () \\ \mathsf{out}_{\mathsf{Seq}}\ (Cons\ (h,t)) = i_2\ (h,t) \end{cases}
\end{array}
$$

resulting in $\mathsf{out}_{\mathsf{Seq}} : A^* \to 1 + A \times A^*$. It is also worth noting that `data Seq a = Nil | Cons (a, Seq a)` is a simplification of `data Seq a = Nil () | Cons (a, Seq a)`. In the latter case, we have $\mathsf{in}_{\mathsf{Seq}} = [Nil, Cons]$, i.e., $Nil : 1 \to A^*$. However, Haskell allows this more "programming-oriented" simplification, where the parameter `()` can be eliminated, resulting in $\mathsf{in}_{\mathsf{Seq}} = [\underline{Nil}, Cons]$.

Note the terminology used: in_{T} and $\mathsf{out}_{\mathsf{T}}$ (in this particular case, $\mathsf{in}_{\mathsf{Seq}}$ and $\mathsf{out}_{\mathsf{Seq}}$ for the type A^*) — in_{T} ("going inside") implies constructing or synthesizing values of A^*, while $\mathsf{out}_{\mathsf{T}}$ ("going outside") suggests deconstructing or analyzing values of A^*. We will frequently rely on this duality throughout the notes.

Since it is not our intention to define an already well-implemented datatype, the objective now is to transpose what we have just concluded to the type `[a]`. By definition, it only requires making small renamings:

$$
A^* \underset{[\underline{[\,]}, \widehat{(:)}]}{\overset{\mathsf{out}_{\mathsf{List}}}{\cong}} 1 + A \times A^* \tag{2.1}
$$

As expected, the function $\mathsf{out}_{\mathsf{List}}$ will be similar to $\mathsf{out}_{\mathsf{Seq}}$:

$$
\begin{array}{ll}
& \mathsf{out}_{\mathsf{List}} \cdot \mathsf{in}_{\mathsf{List}} = id \\
\equiv & \quad \{ \text{ +-fusion (1.28); +-reflection (1.34) } \} \\
& [\mathsf{out}_{\mathsf{List}} \cdot \underline{[\,]}, \mathsf{out}_{\mathsf{List}} \cdot \widehat{(:)}] = [i_1, i_2] \\
\equiv & \quad \{ \text{ +-eq (??); extensional equality (1.2); constant (1.8) } \} \\
& \begin{cases} \mathsf{out}_{\mathsf{List}}\ [\,] = i_1\ () \\ \mathsf{out}_{\mathsf{List}}\ (h : t) = i_2\ (h,t) \end{cases}
\end{array}
$$

In order to simplify the notation, we define the "constructors" $nil = \underline{[\,]}$ and $cons = \widehat{(:)}$. These functions are defined in the **Cp.hs** and **List.hs** libraries of the discipline.

```
1 nil = const []
2
3 cons = uncurry (:)
4
5 inList = either nil cons
6
```

```
7 outList []     = i1 ()
8 outList (a:x) = i2 (a,x)
```

Everything we have just considered pertains to a particular isomorphism, that of constructing finite sequences. However, there are other datatypes, such as trees, hash tables, etc., that our notation and method should be able to accommodate. To achieve this, let us start by generalizing the concept of inductive types. The isomorphism (2.1) can be rewritten as follows:

$$A^* \cong \mathsf{F}\, A^*$$

by introducing the datatype operator $\mathsf{F}\, X = 1 + A \times X$. The operator F defines the recursive pattern associated with the type A^*. This operator is known as a functor and will be the subject of study in the next section. The rule is, given the definition of an inductive type in the form:

$$\mathsf{T} \underset{intT}{\overset{out_{\mathsf{T}}}{\cong}} \mathsf{F}\, \mathsf{T} \tag{2.2}$$

the recursion pattern is dictated by the functor F. For example, $\mathsf{F}\, X = A + X^2$ dictates the recursive pattern of Leaf Trees, where the leaves are elements of type A.

```
data LTree a = Leaf a | Fork (LTree a, LTree a)
```

Renaming T to LTree A,

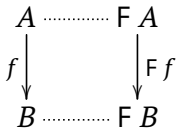
$$\mathsf{LTree}\ A \underset{[Leaf\,,Fork]}{\overset{out_{\mathsf{LTree}}}{\cong}} A + \mathsf{LTree}^2\ A$$

2.3 Functors

A functor F can be seen as a datatype constructor that, given a type A, constructs a more elaborate datatype $\mathsf{F}\ A$. Similarly, given another type B, it constructs, in a similar fashion, a datatype $\mathsf{F}\ B$, and so on.

What is particularly relevant is that the structural effect created by the functor is also extended to functions. Given a function $f : A \to B$, note that the input type and the output type

are parameters of F A and F B, respectively. By definition, if F is a functor, then F f :F $A \to$ F B exists for every f:



F f extends f to structures dictated by the functor F and, by definition, obeys the following two properties:

- **F-functor**:

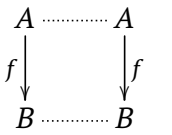
$$\mathsf{F}\ (g \cdot h) = \mathsf{F}\ g \cdot \mathsf{F}\ h \tag{2.3}$$

- **F-id-functor**:

$$\mathsf{F}\ id_A = id_{(\mathsf{F}A)} \tag{2.4}$$

Two basic examples are followed:

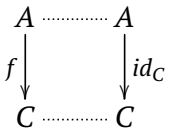
- **Identity functor**:



$$\mathsf{F}\ X = X \qquad \mathsf{F}\ f = id \tag{2.5}$$

  Properties (2.3) and (2.4) are trivially satisfied by simply removing the F symbol.

- **Constant functor**:



$$\mathsf{F}\ X = C \qquad \mathsf{F}\ f = id_C \tag{2.6}$$

  Again, properties (2.3) and (2.4) are trivially satisfied.

Just as functions can be unary, binary, ternary, and so on, functors can also have different arities. A s expected, properties (2.3) and (2.4) should hold for each argument of the n-ary functor. For example, for a binary functor (also known as a bifunctor), equation (2.3) becomes:

$$\mathsf{B}\ (g \cdot h, i \cdot j) = \mathsf{B}\ (g, i) \cdot \mathsf{B}\ (h, j) \tag{2.7}$$

and equation (2.4) becomes

$$\mathsf{B}\ (id_A, id_B) = id_{\mathsf{B}(A,B)} \tag{2.8}$$

Finally, just like functions, functors can also be composed with each other:

$$(\mathsf{F} \cdot \mathsf{G})\ X \stackrel{\text{def}}{=} \mathsf{F}\ (\mathsf{G}\ X) \tag{2.9}$$

$$(\mathsf{F} \cdot \mathsf{G})\ f \stackrel{\text{def}}{=} \mathsf{F}\ (\mathsf{G}\ f) \tag{2.10}$$

2.4 Polynomial functors

Polynomial functors are described by polynomial expressions, such as:

$$\mathsf{F}\ X = 1 + A \times X$$

Therefore, a polynomial functor can be one of the following three cases:

- the identity functor or a constant functor;
- the (finite) product or coproduct of polynomial functors;
- the composition of other polynomial functors.

As mentioned in the previous section, the structural effect of a functor is extended to functions whose types are parameters of the functor. In the case of polynomial functors, this effect is easily obtained when unfolding the functor into the product and/or coproduct of the two basic functors (identity and constant functores). For example, $\mathsf{F}\ X = 1 + A \times X$ can be unfolded into $\mathsf{F}\ X = \mathsf{F}_1\ X + \mathsf{F}_2\ X \times \mathsf{F}_3\ X$, where $\mathsf{F}_1\ X = 1$, $\mathsf{F}_2\ X = A$, and $\mathsf{F}_3\ X = X$. Thus,

$$\begin{array}{ll} & \mathsf{F}_1\ f + \mathsf{F}_2\ f \times \mathsf{F}_3\ f \\ = & \quad \{\ \text{constant functor (2.6) 2x e identity functor (2.5)}\ \} \\ & id_1 + id_A \times f \\ = & \quad \{\ \text{omit subscripts}\ \} \\ & id + id \times f \end{array}$$

Therefore, $1 + A \times X$ denotes the same as $\mathsf{F}\ f = id_1 + id_A \times f$ or even $id + id \times f$ if we omit the subscripts. The idea is that, when constructing the new type $1 + A \times X$ from the type X, the function used to traverse this new structure will be $id + id \times f$, assuming that f is a function that operates on X and that we do not want to modify the other elements of the new structure. Thus, we can think of a functor as a "pair of functions": $\mathsf{F}\ X$ constructs the new datatype, and $\mathsf{F}\ f$ operates on that new type or structure.

Finally, it is important to know that any polynomial functor can be rewritten in canonical form,

$$\begin{array}{l} \mathsf{F}\ X \cong C_0 + (C_1 \times X) + (C_2 \times X^2) + \ldots + (C_n \times X^n) \\ \quad = \sum_{i=0}^{n} C_i \times X^i \end{array} \tag{2.11}$$

and that the Newton's binomial formula

$$(A + B)^n \cong \sum_{p=0}^{n} \binom{n}{p} \times A^{n-p} \times B^p \tag{2.12}$$

can be used for such conversions.

2.5 Polynomial inductive datatypes

An inductive datatype is called polynomial whenever its recursive pattern is described by a polynomial functor, i.e., whenever F in (2.2) is a polynomial.

Polynomial types are easy to define in Haskell whenever the associated functor is in canonical form. Given a type

$$\mathsf{T} \underset{\mathsf{in}_\mathsf{T}}{\cong} \sum_{i=0}^{n} C_i \times \mathsf{T}^i \tag{2.13}$$

one has that

$$\mathsf{in}_\mathsf{T} \stackrel{\text{def}}{=} [\mathsf{T}C_0, \ldots, \mathsf{T}C_n]^1 \tag{2.14}$$

where, for $i \in \{0, \ldots, n\}$, $\mathsf{T}C_i$ is the **T**ype **C**onstructor whose signature is $\mathsf{T} \leftarrow C_i \times \mathsf{T}^i$. In Haskell, the definition of T corresponds to

```
data T = TC0 C0 | TC1 (C1, T) | TC2 (C2, (T,T)) | ... | TPN (CN, (T, ..., T))
```

2.6 Catamorphisms

Given an inductive datatype T, the objective is to express a particular analysis of T, i.e., a function with the signature $f : \mathsf{T} \to B$, for some output type B, for instance, calculating the sum of all elements within a list of naturals. Thus, f corresponds to $sum : \mathbb{N}_0{}^* \to \mathbb{N}_0$. For this purpose, two functions are required,

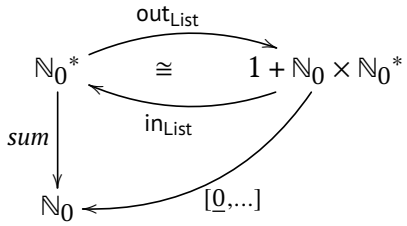
- addition in $\mathbb{N}_0$:

$$\begin{aligned} &add : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\ &add\ (x, y) \stackrel{\text{def}}{=} x + y \end{aligned} \tag{2.15}$$

- addition of "no elements", given by the function $\mathbb{N}_0 \xleftarrow{\underline{0}} 1$, meaning that adding no elements results in 0.

[1] Note the abuse of notation — *either* is a binary operator, such as coproduct, thus functions should be grouped into another alternatives, for instance, $[\mathsf{T}C_0, \mathsf{T}C_1, \mathsf{T}C_2] : \mathsf{T} \leftarrow (C_0 + C_1 \times \mathsf{T} + C_2 \times \mathsf{T}^2)$ should be $[\mathsf{T}C_0, [\mathsf{T}C_1, \mathsf{T}C_2]] : \mathsf{T} \leftarrow (C_0 + (C_1 \times \mathsf{T} + C_2 \times \mathsf{T}^2))$

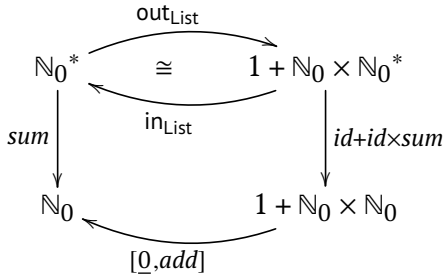
We quickly identify that the function $\underline{0}$ is required for the "base case" in the inductive definition of a list, that is, when the list is empty. Thus, we would have something like this:



And what can be said about the sum of a list given by the construction of a pair head and tail? It is simply the addition of the head with the sum of the tail.

$$\mathbb{N}_0 \times \mathbb{N}_0{}^* \xrightarrow{id+sum} \mathbb{N}_0 \times \mathbb{N}_0 \xrightarrow{add} \mathbb{N}_0$$

Thus, we obtain the following diagram:



(2.16)

which presents the definition of sum:

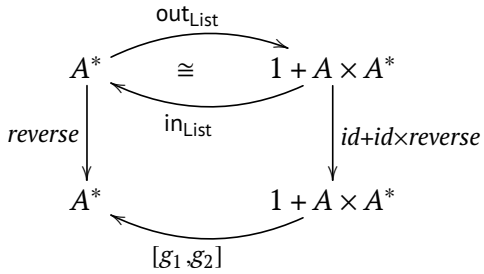
$$sum = [\underline{0}, add] \cdot (id + id \times sum) \cdot \mathsf{out}_{\mathsf{List}} \tag{2.17}$$

easily converted to a pointwise definition:

$$\begin{array}{ll}
 & sum = [\underline{0}, add] \cdot (id + id \times sum) \cdot \mathsf{out}_{\mathsf{List}} \\
\equiv & \quad \{ \text{ shunting rules (1.12) } \} \\
 & sum \cdot \mathsf{in}_{\mathsf{List}} = [\underline{0}, add] \cdot (id + id \times sum) \\
\equiv & \quad \{ \ \mathsf{in}_{\mathsf{List}} \text{ definition; +-fusion (1.28); +-absorption (1.29); +-eq (??) } \} \\
 & \left\{ \begin{array}{l} sum \cdot nil = \underline{0} \\ sum \cdot cons = add \cdot (id \times sum) \end{array} \right. \\
\equiv & \quad \{ \text{ extensional equality (1.2); composition (1.3); constant (1.8); product (1.17) } \} \\
 & \left\{ \begin{array}{l} sum\ [\,] = 0 \\ sum\ (h : t) = h + sum\ t \end{array} \right.
\end{array}$$

Let us now do another analysis or observation on the list datatype. This time, we want to reverse a list. Thus, this particular observation is the function $reverse : A^* \to A^*$ (where the datatype is now parametric). For that, we may question if we can reuse something of the

previous calculation since it is the the same datatype and we may think in the same recursive pattern — reverse of [ ] is also a specific value, this time [ ], and reverse of $(h:t)$ is recursively applying function *reverse* to the tail t and then perform an operation on it with the head, in this case concatenating h at the end. Let us then try to redraw diagram (2.16) in order to fit this new function:

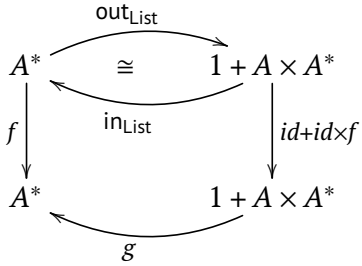


(2.18)

Clearly, $g_1 = \underline{[\,]}$. What about g_2? Well, since g_2 deals with the case of a list by construction $(h:t)$, it will receive the pair $(h, reverse\ t)$. Therefore, $g_2\ (x,y) = y +\!\!+ [x]$.

2.6.1 Introducing catamorphisms over lists

It is worth noting that both functions share a recursive pattern which is very common: when the list is empty they return a specific value. Otherwise, when the list in not empty, they start by recursively invoking the function on the tail whose result is later combined with the head producing the final result — note how similiar both diagrams (2.16) and (2.18) are, only differing in the so-called *gene*, that is, function g in the following general diagram:



(2.19)

Following the standard terminology, we say that f is the List-*catamorphism* induced by g and use the notation $f = (\!|g|\!)$ to express that fact. This terminology is derived from the Greek word $\kappa\alpha\tau\alpha$ (cata) meaning "downwards" — note the direction of the arrow, it is downwards from the inductive type, expressing that a catamorphism allows the transformation of an inductive datatype into any other type in a "destructive" process and dictated by the datatype recursive pattern. Section 2.6 will delve into this subject. For now, it suffices to say that the List-catamorphism induced by $g : 1 + A \times A^* \to B$ is the unique function $(\!|g|\!) : A^* \to B$ defined by

$$(\!|g|\!) = g \cdot (id + id \times (\!|g|\!)) \cdot \mathsf{out}_{\mathsf{List}} \tag{2.20}$$

which is equivalent to

$$(\!|g|\!) \cdot \mathsf{in}_{\mathsf{List}} = g \cdot (id + id \times (\!|g|\!)) \tag{2.21}$$

Besides, note that the Haskell Standard Prelude already has a predefined function for List-catamorphisms. Function *foldr* instantiated for lists follows as

```
foldr           :: (a -> b -> b) -> b -> [a] -> b
foldr f z []    = z
foldr f z (h:t) = f h (foldr f z t)
```

and corresponds to a slightly different implementation of the List-catamorphism — in the case of an empty list, the value is immediately delivered, there is no need for a constant function; in the case of a list by construction, the corresponding function is curried, which means that

$$foldr\ g\ k = (\!|\ [\underline{k}, \widehat{g}]\ |\!) \tag{2.22}$$

For instance,

```
> sum = foldr (+) 0
> sum []
0
> sum [1..4]
10
>
> sum' = cataList (either (const 0) add)
> sum' []
0
> sum' [1..4]
10
```

2.6.2 Generalizing to F-catamorphisms

Given a functor F, any arrow $A \xleftarrow{\alpha} \mathsf{F}\ A$ is said to be an F-algebra, where A is called the *carrier* of the F-algebra α and contains the values that α operates on. This results in the computation of new A-values based on existing ones which are "encapsulated" in a F-pattern structure. Furthermore, given a function $B \xleftarrow{f} A$ and another F-algebra $B \xleftarrow{\beta} \mathsf{F}\ B$, one may consider to relate the F-algebra α to the other F-algebra β in the following manner:

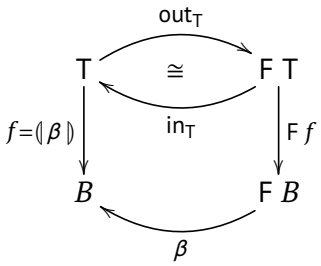
$$\begin{array}{ccc} A & \xleftarrow{\alpha} & \mathsf{F}\ A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\ f} \\ B & \xleftarrow[\beta]{} & \mathsf{F}\ B \end{array} \qquad f \cdot \alpha = \beta \cdot (\mathsf{F}\ f) \tag{2.23}$$

This states that A-objects are mapped to B-objects in a structural way, according to the F-pattern. Arrows with this structure are usually referred to as *homomorphisms*.

It is relevant to this topic the situation where α is an isomorphism (or a bijective function), i.e., that exists some function α° such that $\alpha^{\circ} \cdot \alpha = id$ and $\alpha \cdot \alpha^{\circ} = id$. Such algebras α

are said to be *initial* and usually denoted by in_T, that is, $\mathsf{F}\ \mathsf{T} \xrightarrow{\mathsf{in}_\mathsf{T}} \mathsf{T}$ assuming their carrier set denoted by T. Besides, the converse of the algebra in_T is called the coalgebra out_T. An F-coalgebra is an arrow $\mathsf{F}\ A \longleftarrow A$, for a functor F, where A is also called the *carrier*.

In this particular case, α is such that, for every β, f is unique. The uniqueness of f is denoted by the banana-brackets notation, $f = (\!|\beta|\!)$, followed by the universal property:



$$f = (\!|\beta|\!) \ \Leftrightarrow\ f \cdot \mathsf{in}_\mathsf{T} = \beta \cdot \mathsf{F}\, f \tag{2.24}$$

and so defined as

$$(\!|\beta|\!) \stackrel{\text{def}}{=} \beta \cdot \mathsf{F}\,(\!|\beta|\!) \cdot \mathsf{out}_\mathsf{T} \tag{2.25}$$

$(\!|\beta|\!)$ is referred to as *the (unique) catamorphism* induced by algebra β (or *fold* over β). This construct is a generic and recursive expression that transforms T into B, following a "recursive-descent" approach as dictated by functor F.

As expected, the universal property (2.24) gives rise to the following derived properties:

- **Cata-cancellation**

$$(\!|\alpha|\!) \cdot \mathsf{in}_\mathsf{T} = \alpha \cdot \mathsf{F}\,(\!|\alpha|\!) \tag{2.26}$$

- **Cata-fusion**

$$f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \Leftarrow f \cdot \alpha = \beta \cdot \mathsf{F}\, f \tag{2.27}$$

- **Cata-reflection**

$$(\!|\mathsf{in}_\mathsf{T}|\!) = id_\mathsf{T} \tag{2.28}$$

2.6.3 Catamorphisms over leaf trees

```
data LTree a = Leaf a | Fork (LTree a, LTree a) deriving (Show, Eq, Ord)

inLTree :: Either a (LTree a, LTree a) -> LTree a
inLTree = either Leaf Fork

outLTree                :: LTree a -> Either a (LTree a,LTree a)
outLTree (Leaf a)       = i1 a
outLTree (Fork (t1,t2)) = i2 (t1,t2)
```

2.7 Parameterization and type functors

TBC

Cata-map-definition

$$\mathsf{T}\, f \stackrel{\text{def}}{=} (\!|\, \mathsf{in}_{\mathsf{T}} \cdot \mathsf{B}\ (f, id)\, |\!) \tag{2.29}$$

Cata-absorption

$$(\!|\, g \,|\!) \cdot (\!|\, \mathsf{in}_{\mathsf{T}} \cdot \mathsf{B}\ (f, id)\, |\!) = (\!|\, g \cdot \mathsf{B}\ (f, id)\, |\!) \tag{2.30}$$

2.7.1 Leaf trees' type functor

```
instance Functor LTree
    where fmap f = cataLTree (inLTree . baseLTree f id)
```

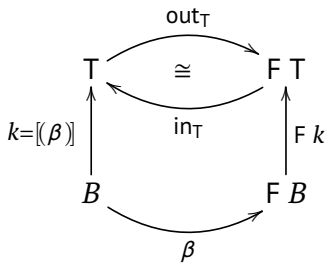
2.8 Anamorphisms

The objective now is to synthetize or produce an inductive datatpye, that is, a function with the signature $f : A \to \mathsf{T}$.

2.8.1 Introducing anamorphisms over lists

Greek $\alpha\nu\alpha$ (ana) meaning "upwards".

2.8.2 Generalizing to F-anamorphisms



$$k = [\!(\beta)\!] \;\Leftrightarrow\; \mathsf{out}_\mathsf{T} \cdot k = (\mathsf{F}\ k) \cdot \beta \tag{2.31}$$

- **Ana-cancellation**

$$out \cdot [\!(g)\!] = \mathsf{F}\ [\!(g)\!] \cdot g \tag{2.32}$$

- **Ana-fusion**

$$[\!(g)\!] \cdot f = [\!(h)\!] \Leftarrow g \cdot f = (\mathsf{F}\ f) \cdot h \tag{2.33}$$

- **Ana-reflection**

$$[\!(\mathsf{out}_\mathsf{T})\!] = id_\mathsf{T} \tag{2.34}$$

- **Ana-map-definition**

$$\mathsf{T}\ f = [\!(\mathsf{B}\ (f, id) \cdot \mathsf{out}_\mathsf{T})\!] \tag{2.35}$$

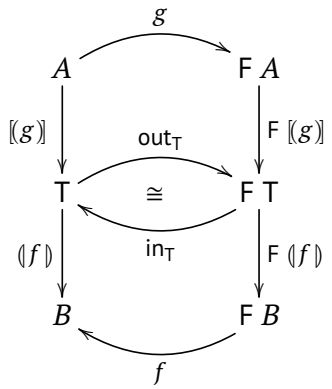
- **Ana-absorption**

$$\mathsf{T}\ f \cdot [\!(g)\!] = [\!(\mathsf{B}\ (f, id) \cdot g)\!] \tag{2.36}$$

2.8.3 Anamorphisms over leaf trees

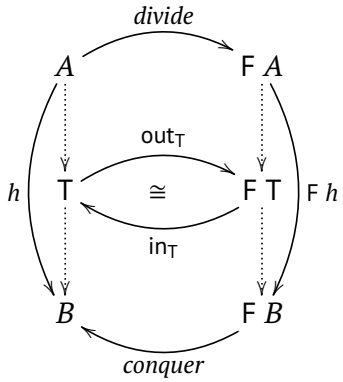
TBC

2.9 Hylomorphisms



$$[\![f, g]\!] = (\!| f |\!) \cdot [\!(g)\!] \tag{2.37}$$

2.9.1 Divide & conquer



$$\begin{aligned} h &= conquer \cdot \mathsf{F}\ h \cdot divide \\ &= (\!| conquer |\!) \cdot [\!(divide)\!] \end{aligned} \tag{2.38}$$

Problem 4 Consider the following system of maintaining and using a dictionary: it will be built over a tree in which each node will have only one letter of the word and each leaf the respective translation (one or more synonyms).

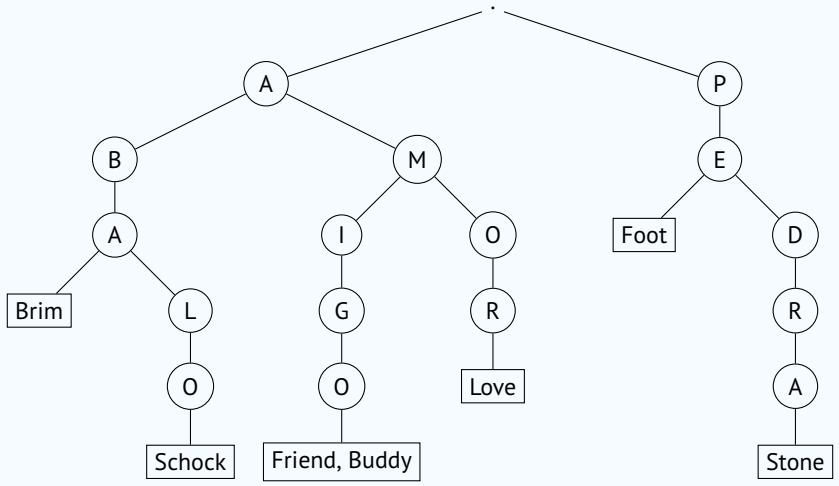


Fig. 1: Example of a dictionary.

```
data Exp v o =   Var v                -- expressions are either variables
               | Term o [ Exp v o ] -- or terms involving operators and
                                     -- subterms
               deriving (Show,Eq)
type Dictionary = Exp [String] Char
```

Implement the operation `translate :: (String, Dictionary) -> [String]` that searches for translations for a given word, using the *divide & conquer* method.

Solution

$$\begin{array}{ccccc} S \times D & \xrightarrow{\quad divide = \phi \cdot (\mathrm{out}_{List} \times \mathrm{out}_{Exp}) \quad} & & & 1 + (S^* + (S \times D)^*) \\ & \searrow & (1 + C \times S) \times (S^* + C \times D^*) & \xrightarrow{\phi} & \\ \Big\downarrow translate & & & & \Big\downarrow id + (id + \mathrm{map}\ translate) \\ S^* & \xleftarrow[conquer = [nil, [id, concat]]]{} & & & 1 + (S^* + (S^*)^*) \end{array}$$

```
translate = conquer . (id -|- (id -|- map translate)) . divide where
    divide = phi . (outList >< outExp)
    phi (Left (), Left ts) = i2 (i1 ts)
    phi (Left (), Right _) = i1 ()
    phi (Right _, Left _) = i1 ()
    phi (Right (c1, cs), Right (c2, ds)) =
        if (c2 == '.') then (i2 . i2 . r) (c1:cs, ds)
                       else if (c1 == c2) then (i2 . i2 . r) (cs, ds)
                                          else i1 ()
    r (s, ds) = map (\d -> (s, d)) ds
    conquer = either nil (either id concat)
```

2.10 Mutual recursion

TBC

The Fibonacci sequence In mathematics, the Fibonacci numbers, usually denoted by F_n, form a sequence of natural numbers, named *Fibonacci sequence*, where each number is obtained from the addition of the previous two, i.e.,

$$
\begin{aligned}
&F_0 = 1\\
&F_1 = 1\\
&F_{n+1} = F_n + F_{n-1},\ n > 1
\end{aligned}
$$

For example, the first twelve elements are, respectively,

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144$$

TBC

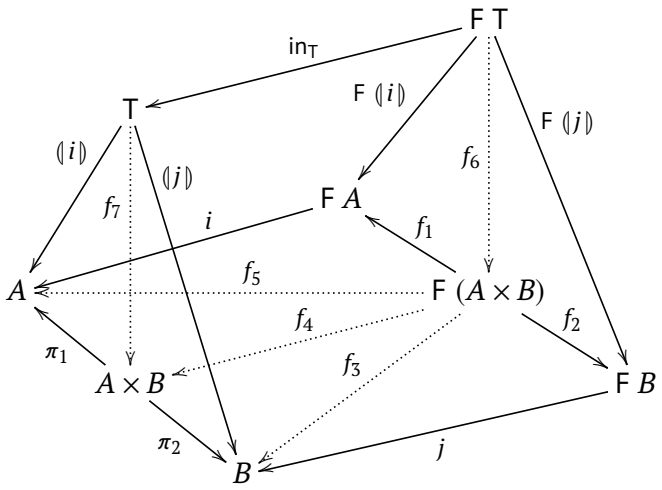
$$
\begin{aligned}
&\left\{\begin{array}{l} f \cdot \mathsf{in}_{\mathsf{T}} = h \cdot \mathsf{F}\ \langle f, g\rangle \\ g \cdot \mathsf{in}_{\mathsf{T}} = k \cdot \mathsf{F}\ \langle f, g\rangle \end{array}\right. \quad \equiv \quad \langle f, g\rangle = (\!|\, \langle h, k\rangle \,|\!) \\
\equiv \quad &\{\ \mathsf{in}_{\mathsf{T}} = [\underline{0}, succ]\ ;\ \mathsf{F}\ f = id + f\ \} \\
&\left\{\begin{array}{l} f \cdot [\underline{0}, succ] = h \cdot (id + \langle f, g\rangle) \\ g \cdot [\underline{0}, succ] = k \cdot (id + \langle f, g\rangle) \end{array}\right. \quad \equiv \quad \langle f, g\rangle = (\!|\, \langle h, k\rangle \,|\!) \\
\equiv \quad &\{\ \text{+-fusion (1.28)}\ ;\ h = [h_1, h_2]\ \mathsf{e}\ k = [k_1, k_2]^{1}\ \} \\
&\left\{\begin{array}{l} [f \cdot \underline{0}, f \cdot succ] = [h_1, h_2] \cdot (id + \langle f, g\rangle) \\ [g \cdot \underline{0}, g \cdot succ] = [k_1, k_2] \cdot (id + \langle f, g\rangle) \end{array}\right. \quad \equiv \quad \langle f, g\rangle = (\!|\, \langle [h_1, h_2], [k_1, k_2]\rangle \,|\!) \\
\equiv \quad &\{\ \text{extensional equality (1.2)}\ ;\ \text{composition (1.3)}\ ;\ \text{split (1.13)}\ ;\ \times\text{-def (1.16)}\ \} \\
&\left\{\begin{array}{l} \left\{\begin{array}{l} f\ 0 = a \\ f\ (n+1) = h_2\ (f\ n, g\ n) \end{array}\right. \\ \left\{\begin{array}{l} g\ 0 = b \\ g\ (n+1) = k_2\ (f\ n, g\ n) \end{array}\right. \end{array}\right. \quad \equiv \quad \langle f, g\rangle = (\!|\, \langle [\underline{a}, h_2], [\underline{b}, k_2]\rangle \,|\!)
\end{aligned}
$$

```
int fibonacci(int n){
    int x = 1, y = 1, i;
```

[1] h and k are post-composed with a coproduct and therefore are alternatives (*eithers*).

```
    for (i = 1; i <= n; i++){
        int a = x;
        x = x + y;
        y = a;
    }
    return y;
}
```

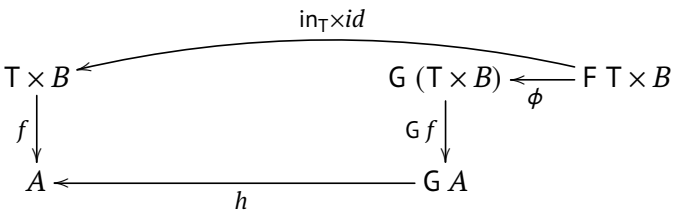
2.11 "Banana-split" — a corollary of the mutual recursion law



2.12 Higher-order catamorphisms

$$f \cdot (\mathsf{in}_{\mathsf{T}} \times id) = h \cdot \mathsf{G}\, f \cdot \phi \quad \Leftrightarrow \quad \overline{f} = (\!|\, \overline{h \cdot \mathsf{G}\, ap \cdot \phi} \,|\!) \tag{2.39}$$

The left-hand side of (2.39) is the G-hylomorphism



$$f = h \cdot \mathsf{G}\, f \cdot \phi \cdot (\mathsf{out}_{\mathsf{T}} \times id)$$

which is equivalent to the F-catamorphism

$$\begin{array}{ccc} \mathsf{T} & \xleftarrow{\mathsf{in}_{\mathsf{T}}} & \mathsf{F}\ \mathsf{T} \\ {\scriptstyle \overline{f}}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\ \overline{f}} \\ A^B & \xleftarrow[\overline{h \cdot G\ ap \cdot \phi}]{} & \mathsf{F}\ A^B \end{array}$$

$$A \xleftarrow{h} \mathsf{G}\ A \xleftarrow{\mathsf{G}\ ap} \mathsf{G}\ (A^B \times B) \xleftarrow{\phi} \mathsf{F}\ A^B \times B$$

$$\overline{f} = (\!|\, \overline{h \cdot G\ ap \cdot \phi} \,|\!)$$

Function *take* as a higher-order catamorphism The *uncurried* version of the function *take*, with signature $\widehat{take} :: \mathbb{N}_0 \times A^* \to A^*$, is given by the following anamorphism on lists:

$$\begin{array}{ccc} \mathbb{N}_0 \times A^* & \xleftarrow{\mathsf{L}\ \mathsf{in}_{\mathbb{N}_0}} & 1 + A \times (\mathbb{N}_0 \times A^*) \xleftarrow{\phi} (1 + \mathbb{N}_0) \times A^* \\ {\scriptstyle \widehat{take}}\downarrow & & \downarrow{\scriptstyle \mathsf{G}\ \widehat{take}} \\ A^* & \xleftarrow[\mathsf{in}_{\mathsf{List}}]{} & 1 + A \times A^* \end{array}$$

where

$$\begin{array}{l} \phi = [i_1 \cdot \pi_1, (\pi_2 + xr) \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})] \cdot distl \\ xr = \langle \pi_1 \cdot \pi_2, id \times \pi_2 \rangle \\ \mathsf{L}\ f = f \times id \\ \mathsf{G}\ f = id + id \times f \end{array}$$

that is,

$$\widehat{take} = [\!(\phi \cdot \mathsf{L}\ \mathsf{out}_{\mathbb{N}_0})\!]$$

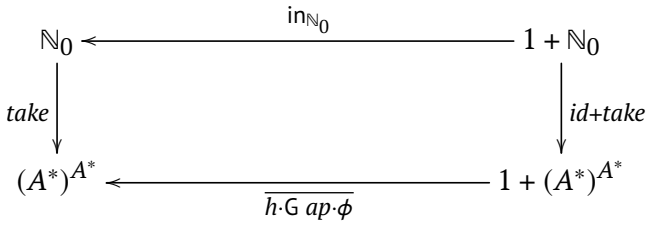
or, from a more general perspective, by the hylomorphism

$$\widehat{take} = [\!\![\, \mathsf{in}_{\mathsf{List}}, \phi \cdot \mathsf{L}\ \mathsf{out}_{\mathbb{N}_0} \,]\!\!]$$

which corresponds to

$$\widehat{take} = \mathsf{in}_{\mathsf{List}} \cdot \mathsf{G}\ \widehat{take} \cdot \phi \cdot \mathsf{L}\ \mathsf{out}_{\mathbb{N}_0}$$

Now, let us recurr to the power of the adjoint recursion in order to define the original (*curried*) version of function *take* as a higher order catamorphism:



$$take = (\!|\, \overline{\mathsf{in}_{\mathsf{List}} \cdot \mathsf{G}\ ap \cdot \phi}\ |\!)$$

Simplifying the gene of the catamorphism:

$$\begin{array}{ll}
& [f,g] = \overline{\mathsf{in}_{\mathsf{List}} \cdot \mathsf{G}\ ap \cdot \phi} \\
= & \quad \{\ \text{Definition of G } ap\ \} \\
& [f,g] = \overline{\mathsf{in}_{\mathsf{List}} \cdot (id + id \times ap) \cdot [i_1 \cdot \pi_1, (\pi_2 + xr) \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})] \cdot distl} \\
= & \quad \{\ \} \\
& [f,g] = \overline{[nil, cons \cdot (id \times ap)] \cdot [i_1 \cdot \pi_1, (\pi_2 + xr) \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})] \cdot distl} \\
= & \quad \{\ \} \\
& [f,g] = \overline{[nil, [nil, cons \cdot (id \times ap)] \cdot (\pi_2 + xr) \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})] \cdot distl} \\
= & \quad \{\ \} \\
& [f,g] = \overline{[nil, [nil, cons \cdot (id \times ap) \cdot xr] \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})] \cdot distl} \\
= & \quad \{\ [\overline{f}, \overline{g}] = \overline{[f,g] \cdot distl}\ \} \\
& \left\{ \begin{array}{l} f = \overline{nil} \\ g = \overline{[nil, cons \cdot (id \times ap) \cdot xr] \cdot distr \cdot (id \times \mathsf{out}_{\mathsf{List}})} \end{array} \right. \\
= & \quad \{\ \overline{f \cdot (id \times g)} = (\cdot g) \cdot \overline{f}\ \} \\
& \left\{ \begin{array}{l} f = \overline{nil} \\ g = (\cdot \mathsf{out}_{\mathsf{List}}) \cdot \overline{[nil, cons \cdot (id \times ap) \cdot xr] \cdot distr} \end{array} \right.
\end{array}$$

In order to continue simplifying the gene, remember that $A \times B \cong B \times A$. Like this, using the function $flip : (A \to B \to C) \to B \to A \to C$, we can easily identify the following isomorphism:

$$K \times A \to B \quad \underset{flip\cdot curry}{\overset{uncurry\cdot flip}{\cong}} \quad A \to B^K \tag{2.40}$$

from which the following property is extracted:

$$[flip\ \overline{f}, flip\ \overline{g}] = flip\ \overline{[f,g] \cdot distr} \tag{2.41}$$

which will be of help in the further simplification:

$$
\begin{cases} f = \overline{nil} \\ g = (\cdot \mathsf{out}_{\mathsf{List}}) \cdot (flip\ (flip\ \overline{[nil, cons \cdot (id \times ap) \cdot xr] \cdot distr})) \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ g = (\cdot \mathsf{out}_{\mathsf{List}}) \cdot (flip\ [flip\ \overline{nil}, flip\ \overline{cons \cdot (id \times ap) \cdot xr}]) \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ flip\ ((\cdot \mathsf{in}_{\mathsf{List}}) \cdot g) = [flip\ \overline{nil}, flip\ \overline{cons \cdot (id \times ap) \cdot xr}] \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ \begin{cases} (flip\ ((\cdot \mathsf{in}_{\mathsf{List}}) \cdot g)) \cdot i_1 = flip\ \overline{nil} \\ (flip\ ((\cdot \mathsf{in}_{\mathsf{List}}) \cdot g)) \cdot i_2 = flip\ \overline{cons \cdot (id \times ap) \cdot xr} \end{cases} \end{cases}
$$

$$
= \quad \{\ flip\text{-fusion ; remove } flip \text{ both sides}\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ \begin{cases} (\cdot i_1) \cdot (\cdot \mathsf{in}_{\mathsf{List}}) \cdot g = \overline{nil} \\ (\cdot i_2) \cdot (\cdot \mathsf{in}_{\mathsf{List}}) \cdot g = \overline{cons \cdot (id \times ap) \cdot xr} \end{cases} \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ \begin{cases} (\cdot (\mathsf{in}_{\mathsf{List}} \cdot i_1)) \cdot g = \overline{nil} \\ (\cdot (\mathsf{in}_{\mathsf{List}} \cdot i_2)) \cdot g = \overline{cons \cdot (id \times ap) \cdot xr} \end{cases} \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ \begin{cases} (g\ k) \cdot nil = nil \\ (g\ k) \cdot cons = \overline{cons \cdot (id \times ap) \cdot xr}\ k \end{cases} \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ \begin{cases} g\ k\ [\,] = [\,] \\ g\ k\ (h : t) = \overline{cons \cdot (id \times ap) \cdot xr}\ k\ (h, t) \end{cases} \end{cases}
$$

$$
= \quad \{\ \}
$$

$$
\begin{cases} f = \overline{nil} \\ \begin{cases} g\ k\ [\,] = [\,] \\ g\ k\ (h : t) = h : k\ t \end{cases} \end{cases}
$$

Thus, function *take* is given by the following (higher order) catamorphism:

$$
\begin{aligned}
&take = (\!|\, [f, g] \,|\!) \ \textbf{where} \\
&\quad f = \overline{nil} \\
&\quad g\ k\ [\,] = [\,] \\
&\quad g\ k\ (h : t) = h : k\ t
\end{aligned}
$$

Chapter 3

Introduction to monadic programming

Monads are race functors

J.N. Oliveira

This chapter introduces a significant tool in modern functional programming, namely the monad. The concept of a monad holds great relevance in computer science today as it serves as a special case of a functor, a "race functor". This name is due to the fact that it provides a unified and elegant approach to describe various computational effects, including input/output, comprehension notation, state variable manipulation, probabilistic behavior, partial behavior, nondeterminism, and more.

We shall start this chapter by analyzing the already mentioned partial behavior — the situation where a function is defined for some inputs but not for others, i.e., a function may produce a valid result for certain inputs but may fail for others — and nondeterminism — in this case, the ability to represent and manipulate multiple possible values or choices (which does not mean that the computation is random or unpredictable in the usual sense of nondeterminism).

3.1 Partial functions

Recall function $head : [a] \to a$ which outputs the first element of a finite list. It is evident that $head\ [\,]$ is undefined since $[\,]$ contains no elements. Thus, attempting to output the first element from it yields an undefined result:

```
> head []
*** Exception: Prelude.head: empty list
```

Functions like *head* are referred to as partial functions because they cannot be applied to all well-typed inputs, that is, they may diverge or produce an error for some of such (well-typed) inputs. The occurrence of this potentially dangerous behavior is a significant concern in programming. To mitigate this behavior, there are two alternatives. The first approach is to ensure that every call to $head\ x$ is protected by verifying the pre-condition $x \neq [\,]$. This involves wrapping such calls in contexts that guarantee the required conditions. The second approach is to handle exceptions explicitly, using, for example, the parametric datatype *Maybe*:

```
head'        :: [a] -> Maybe a
head' []     = Nothing
head' (h:t) = Just x
```

In words, $head'\ x$ maybe return a value — if $x \neq [\,]$ it clearly does, but for $x = [\,]$ it doesn't. Therefore, the *Maybe* datatype may be used to handle with undefined cases. Besides, note that $Maybe\ A \cong 1 + A$ acting like a *pointer* as in e.g. the C programming language. Thus, one may regard as partial every function of type

$$1 + B \xleftarrow{g} A$$

for some A and B, that is, partial functions point to 1 in some cases.

3.2 Composing partial functions

Partial functions do not compose in general:

$$\begin{array}{ccc} & & 1 + B \xleftarrow{g} A \\ & & \vdots \\ 1 + C & \xleftarrow{f} & B \end{array}$$

f needs to be somehow extended to be able to accpet arguments of type $1 + B$:

$$\begin{array}{ccccc} & \overset{i_1}{\nearrow} & 1 + B & \xleftarrow{g} & A \\ 1 & \overset{[i_1, f]}{\searrow} & \vdots & & \\ \downarrow{\scriptstyle i_1} & & \uparrow{\scriptstyle i_2} & & \\ 1 + C & \xleftarrow{f} & B & & \end{array} \tag{3.1}$$

Note how the exception produced by the *producer* function g is propagated to the output of the *consumer* function f. Thus, we may define the composition of partial functions as

$$f \bullet g \overset{\text{def}}{=} [i_1, f] \cdot g \tag{3.2}$$

This means that,

$$
\begin{aligned}
&(f \bullet g)\ a = f'\ (g\ a)\ \textbf{where}\\
&\quad f'\ (i_1\ ()) = i_1\ ()\\
&\quad f'\ (i_2\ x) = f\ x
\end{aligned}
$$

or, in terms of the *Maybe* datatpye,

$$
\begin{aligned}
&(f \bullet g)\ a = f'\ (g\ a)\ \textbf{where}\\
&\quad f'\ \mathit{Nothing} = \mathit{Nothing}\\
&\quad f'\ (\mathit{Just}\ x) = f\ x
\end{aligned}
$$

Now, it is of paramount importance to note that diagram 3.1 can be subject to to redesign when considering that, by +-absorption (1.29), $[i_1, f] = [i_1, id] \cdot (id + f)$:

$$
\begin{array}{ccccc}
1 + (1 + C) & \xleftarrow{id+f} & 1 + B & \xleftarrow{g} & A \\
{\scriptstyle [i_1, id]}\downarrow & & \vdots & & \\
1 + C & \xleftarrow{\quad f \quad} & B & &
\end{array}
$$

The significant importance relies on the fact that it can be simplified when considering functor

$$
\begin{cases}
\mathsf{F}\ X = 1 + X \\
\mathsf{F}\ f = id + f
\end{cases}
$$

and assuming $\mu = [i_1, id]$, that is,

$$
\begin{array}{ccccc}
\mathsf{F}\ (\mathsf{F}\ C) & \xleftarrow{\mathsf{F}\ f} & \mathsf{F}\ B & \xleftarrow{g} & A \\
{\scriptstyle \mu}\downarrow & & \vdots & & \\
\mathsf{F}\ C & \xleftarrow{\quad f \quad} & B & &
\end{array}
\tag{3.3}
$$

Before explaining the reason behind this, let us consider another example.

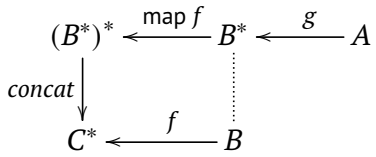
3.3 List nondeterminism

Another computational effect is that of producing multiple outputs, which can be encapsulated, for example, within a list:

$$
\begin{array}{ccc}
 & & B^* \\
 & & \vdots \\
C^* & \xleftarrow{\quad f \quad} & B
\end{array}
\quad
\begin{array}{c}
\xleftarrow{g} \\
\\
A
\end{array}
$$

In this case, the consumer function f must be applied to all those elements produced by the producer function g and we may consider that the output values should be concatenated.

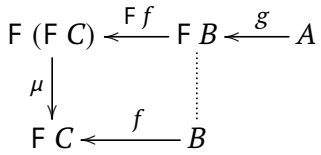
That is,



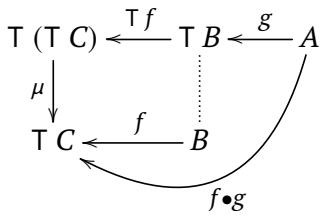
Again, and this time considering functor

$$\begin{cases} \mathsf{F}\ X = X^* \\ \mathsf{F}\ f = \mathsf{map}\ f \end{cases}$$

and $\mu = concat$, one obtains exactly the same diagram:

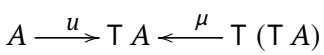


3.4 Finally monads!



(3.4)

$$f \bullet g \stackrel{\text{def}}{=} \mu \cdot \mathsf{T}\ f \cdot g \tag{3.5}$$



3.4.1 Monad LTree

Recall the Haskell definition of a Leaf Tree:

```
data LTree a = Leaf a | Fork (LTree a, LTree a)
```

This datatpye leads to the following isomorphism:

$$\text{LTree } A \underset{[Leaf\,,Fork]}{\overset{\text{out}_{\text{LTree}}}{\cong}} A + (\text{LTree } A)^2$$

As we have seen, a monad has two operations at its disposal, $u : A \to \mathsf{T}\ A$ and $\mu : \mathsf{T}\ (\mathsf{T}\ A) \to A$, which in this case are clearly identified as

$$A \xrightarrow{\ Leaf\ } \text{LTree } A \xleftarrow{\ (\!|\,[id,Fork]\,|\!)\ } \text{LTree } (\text{LTree } A)$$

Let us know check if properties *unit* (**??**) and *multiplication* (**??**) hold:

- *unit*: $\mu \cdot u = \mu \cdot T\ u = id$

$$\begin{array}{ll} & \mu \cdot u \\ = & \quad \{\ \mu = (\!|\,[id, Fork]\,|\!);\ u = Leaf\ \} \\ & (\!|\,[id, Fork]\,|\!) \cdot Leaf \\ = & \quad \{\ \text{cata-definition (2.25)};\ (\!|\,[id, Fork]\,|\!) = \mu\ \} \\ & [id, Fork] \cdot (id + \mu^2) \cdot \text{out}_{\text{LTree}} \cdot Leaf \\ = & \quad \{\ \text{out}_{\text{LTree}} \cdot Leaf = i_1 \text{ since } Leaf = \text{in}_{\text{LTree}} \cdot i_1\ \} \\ & [id, Fork] \cdot (id + \mu^2) \cdot i_1 \\ = & \quad \{\ i_1\text{-natural (1.30)};\ id\text{-natural (1.7)}\ \} \\ & [id, Fork] \cdot i_1 \\ = & \quad \{\ +\text{-cancellation (1.27)}\ \} \\ & id \\ \square & \end{array}$$

$$\begin{array}{ll} & \mu \cdot \mathsf{T}\ u \\ = & \quad \{\ \mu = (\!|\,[id, Fork]\,|\!);\ u = Leaf;\ \mathsf{T} = \text{LTree}\ \} \\ & (\!|\,[id, Fork]\,|\!) \cdot \text{LTree } Leaf \\ = & \quad \{\ \text{cata-absorption (2.30)}\ \} \\ & (\!|\,[id, Fork] \cdot (Leaf + id^2)\,|\!) \\ = & \quad \{\ +\text{-absorption (1.29)};\ \times\text{-id-functor (1.23)};\ \text{id-natural (1.7)}\ \} \\ & (\!|\,[Leaf\,, Fork]\,|\!) \\ = & \quad \{\ \text{cata-reflection (1.34)}\ \} \\ & id \\ \square & \end{array}$$

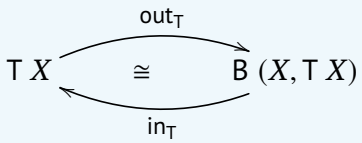
- *multiplication*: $\mu \cdot \mu = \mu \cdot T\ \mu$

$$
\begin{array}{ll}
 & \mu \cdot \mu = \mu \cdot T\ \mu \\
\equiv & \quad \{\ \mu = (\!|\, [id, Fork] \,|\!);\ \text{cata-absorption (2.30)}\ \} \\
 & \mu \cdot (\!|\, [id, Fork] \,|\!) = (\!|\, [id, Fork] \cdot (\mu + \mathsf{F}\ id) \,|\!) \\
\equiv & \quad \{\ \text{+-absorption (1.29); id-functor (2.5); } id\text{-natural (1.7)}\ \} \\
 & \mu \cdot (\!|\, [id, Fork] \,|\!) = (\!|\, \mu\ Fork \,|\!) \\
\Leftarrow & \quad \{\ \text{cata-fusion (??)}\ \} \\
 & \mu \cdot [id, Fork] = [\mu, Fork] \cdot (id + \mu^2) \\
\equiv & \quad \{\ \} \\
 & \left\{ \begin{array}{l} \mu = \mu \\ (\!|\, [id, Fork] \,|\!) \cdot Fork = Fork \cdot \mu^2 \end{array} \right. \\
\equiv & \quad \{\ \text{trivial; } Fork = \mathsf{in}_{\mathsf{LTree}} \cdot i_2\ \} \\
 & (\!|\, [id, \mathsf{in}_{\mathsf{LTree}} \cdot i_2] \,|\!) \cdot \mathsf{in}_{\mathsf{LTree}} \cdot i_2 = \mathsf{in}_{\mathsf{LTree}} \cdot i_2 \cdot \mu^2 \\
\equiv & \quad \{\ \text{cata-cancellation (??)}\ \} \\
 & [id, \mathsf{in}_{\mathsf{LTree}} \cdot i_2] \cdot (id + \mu^2) \cdot i_2 = \mathsf{in}_{\mathsf{LTree}} \cdot i_2 \cdot \mu^2 \\
\equiv & \quad \{\ \text{+-absorption (1.29)}\ \} \\
 & [id, \mathsf{in}_{\mathsf{LTree}} \cdot i_2 \cdot (sqm\ \mu)] = \mathsf{in}_{\mathsf{LTree}} \cdot i_2 \cdot \mu^2 \\
\equiv & \quad \{\ \text{+-cancellation (1.27)}\ \} \\
 & true \\
\Box &
\end{array}
$$

Thus, we may define an instance of the class **Monad** as

```
instance Monad LTree where
    return  = Leaf
    t >>= g = (mu . fmap g) t where
       mu  =  cataLTree (either id Fork)
```

The monad LTree is a particular case of the following monad:



$$
\begin{cases} \mathsf{B}\ (X, Y) = X + \mathsf{F}\ Y \\ \mathsf{B}\ (f, g) = f + \mathsf{F}\ g \end{cases} \quad \text{for a given functor F}
$$

$$\mu = (\!|\, [id, \mathsf{in}_{\mathsf{T}} \cdot i_2] \,|\!)$$

$$u = \mathsf{in}_{\mathsf{T}} \cdot i_1$$

3.5 Monadic application (or binding)

TBC

3.6 Sequencing and the do-notation

TBC

3.7 Monadic recursion

TBC

3.8 The state monad

TBC

3.9 The monad IO

TBC

Problem 5 The Truchet tiles are patterns obtained by randomly generating two-dimensional combinations of basic tiles. Those shown in the figure 2 are known as Truchet-Smith tiles

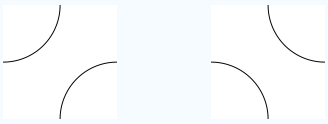


Fig. 2: Truchet-Smith tiles.

and may be displayed in Haskell with the Gloss library as

```
put = uncurry Translate

truchet1 =
    Pictures [put (0,80) (Arc (-90) 0 40), put (80,0) (Arc 90 180 40)]

truchet2 =
    Pictures [put (0,0) (Arc 0 90 40), put (80,80) (Arc 180 (-90) 40)]
```

Note that the tiles have dimensions 80x80. In this problem it is intend to program the random generation of mosaics of Truchet-Smith tiles using the Random monad and the Gloss library to produce the result. Besides, the program should generate mosaics of any size.

Solution

```
pickTruchet :: R Picture
pickTruchet = pick $ Probability.uniform [truchet1, truchet2]

mosaic x y = splitIn x $ take (x*y) $ repeat pickTruchet where
    splitIn n = anaList ((((!) -|- (splitAt n . cons)) . outList)
    -- splitIn n [] = []
    -- splitIn n t = (cons . (id >< splitIn n) . splitAt n) t

translateX _ _ _ [] = []
translateX n i j [h] = [put (80*i, 0) h]
translateX n i j (h1:h2:t) =
    put (i*80, 0) h1 : put (j*80, 0) h2 : translateX (n-2) (i+1) (j-1) t

translateY _ _ _ [] = []
translateY n i j [h] = [map (put (0, 80*i)) h]
translateY n i j (h1:h2:t) =
    map (put (0, i*80)) h1 : map (put (0, j*80)) h2 :
        translateY (n-2) (i+1) (j-1) t

translateTiles x = translateY x 0 (-1) . map (translateX x 0 (-1))

displayTruchet :: Int -> Int -> IO ()
displayTruchet x y = do
    m <- sequence (map sequence (mosaic x y))
    let window = InWindow
             "Truchet"      -- window title
             (80*x, 80*y)   -- window size
             (0, 0)         -- window position
    display window white ((pictures . concat . (translateTiles x)) m)
```

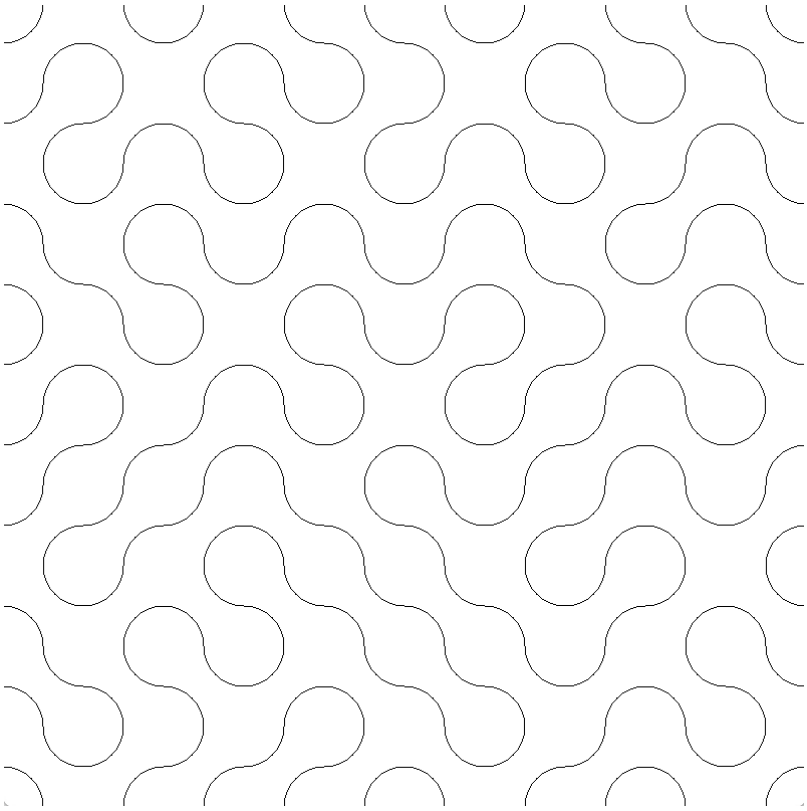


Fig. 3: Output of `displayTruchet 10 10`.

3.10 The Adventurers' Problem

> *In the middle of the night, four adventurers encounter a shabby rope-bridge spanning a deep ravine. For safety reasons, they decide that no more than 2 people should cross the bridge at the same time and that a flashlight needs to be carried by one of them in every crossing. They have only one flashlight. The 4 adventurers are not equally skilled: crossing the bridge takes them 1, 2, 5, and 10 minutes, respectively. A pair of adventurers crosses the bridge in an amount of time equal to that of the slowest of the two adventurers.*
>
> *One of the adventurers claims that they cannot be all on the other side in less than 19 minutes. One companion disagrees and claims that it can be done in 17 minutes with just 5 moves. Who is right? That's what we're going to find out.*

The solution is to take advantage of the non-deterministic monad (`List`) to use brute force and calculate all possible moves until we reach the final state. To deal with the time adventurers need to cross, we'll use the duration monad (implemented by prof. Renato Neves in is lectures of Cyber-Physical Programming [Neves]) which adds the time each adventurer takes in a given move.

```haskell
data Duration a = Duration (Int, a) deriving Show

getDuration :: Duration a -> Int
getDuration (Duration (d, x)) = d

getValue :: Duration a -> a
getValue (Duration (d, x)) = x

instance Functor Duration where
  fmap f (Duration (i, x)) = Duration (i, f x)

instance Applicative Duration where
  pure x = (Duration (0, x))
  (Duration (i, f)) <*> (Duration (j, x)) = (Duration (i+j, f x))

instance Monad Duration where
    (Duration (i, x)) >>= k = Duration (i + getDuration (k x), getValue (k x))
    return = pure
```

This duration monad will be "encapsulated" in our monad **AdventurersLog**. This one will offer, for a certain state, a list of the possible following states with the respective duration needed to get it and will also offer the path traveled at the moment which will be expressed as a *string*, and its elegance will become apparent as we delve into its details. For now, let's analyze the construction of our monad!

```haskell
data AdventurersLog a = ALog [Duration (String, a)] deriving Show

remALog :: AdventurersLog a -> [Duration (String, a)]
remALog (ALog a) = a

instance Functor AdventurersLog where
    fmap f = ALog . map (fmap (id >< f)) . remALog

instance Applicative AdventurersLog where
    pure a = ALog [Duration (0, ("", a))]
    f <*> l = ALog $ do
        Duration (d1, (p1, g)) <- remALog f
        Duration (d2, (p2, a)) <- remALog l
        return (Duration (d1+d2, (p1++p2, g a)))

instance Monad AdventurersLog where
    return = pure
    x >>= f = ALog $ do
        Duration (d, (p, a)) <- remALog x
        map (\ (Duration (d', (p', a))) ->
            (Duration (d + d', (p ++ p', a)))) (remALog (f a))
```

Modelling the problem Adventurers are represented by the following datatype:

```haskell
data Adventurer = P1 | P2 | P5 | P10 deriving (Show, Eq)
```

The names for the adventurers are quite suggestive as they are identified by the time they

take to cross. However, it will be very useful to have a function that returns, for each adventurer, the time it takes to cross the bridge.

```
getTimeAdv :: Adventurer -> Int
getTimeAdv P1 = 1
getTimeAdv P2 = 2
getTimeAdv P5 = 5
getTimeAdv P10 = 10
```

The lantern is represented by the element `()`, so we can represent all the entities by using the coproduct `Either Adventurer ()`. Besides, we need to define the state of the crossing, i.e., the current position of each adventurer (and the lantern too). The function *False* represents the initial state of the crossing, with all adventurers and the lantern on the left side of the bridge. Similarly, the function *True* represents the end state of the crossing, with all adventurers and the lantern on the right side of the bridge. We also need to define the instances *Show* and *Eq* to visualize and compare, respectively, the states of the crossing.

```
type State = Either Adventurer () -> Bool

instance Show State where
    show s = show (show [s (Left P1),
                         s (Left P2),
                         s (Left P5),
                         s (Left P10),
                         s (Right ())])

instance Eq State where
    s1 == s2 = and [s1 (Left P1) == s2 (Left P1),
                    s1 (Left P2) == s2 (Left P2),
                    s1 (Left P5) == s2 (Left P5),
                    s1 (Left P10) == s2 (Left P10),
                    s1 (Right ()) == s2 (Right ())]

cInit :: State
cInit = const False

cEnd :: State
cEnd = const True

state2List :: State -> [Bool]
state2List s = [s (Left P1),
                s (Left P2),
                s (Left P5),
                s (Left P10),
                s (Right ())]
```

Obviously, it is useful a function that changes the state of the crossing for a given object:

```
changeState :: Either Adventurer () -> State -> State
changeState a s = \ a' -> if a' == a then not (s a') else s a'
```

Even more useful is a function that changes the state of the crossing of a list of objects:

```
mChangeState :: [Either Adventurer ()] -> State -> State
mChangeState x s = foldr changeState s x
```

With this, we are now ready to define all the valids plays the adventurers can make for a given state storing the respective duration required and the move made. So, for a given $s : State$, we'll compute $allValidPlays : AdventurersLog\ State \sim LSD\ [Duration\ (String, State)]$. For that, we need to

1. move adventurers — but only adventurers who can pick up the lantern. So, for that given state, we first need to calculate the adventurers who are where the lantern is.

   ```
   advsWhereLanternIs :: State -> [Adventurer]
   advsWhereLanternIs s = filter ((== s (Right ())) . s . Left)
                              [P1, P2, P5, P10]
   ```

2. group them into all possible combinations. As we know, a maximum of 2 adventurers can cross. This parametric function

   ```
   comUpTo2 :: Eq a => [a] -> [[a]]
   comUpTo2 = conc . (split f g) where
         f t = do {x <- t; return [x]}
         g t = do {x <- t; y <- (remove x t); return [x, y]}
         remove x [] = []
         remove x (h:t) = if x==h then t else remove x t
   ```

   applied to the list of all possivle adventurers will return all possible groups in sublists.

3. add the time both group needs to cross — we just need to map the function *getTimeAdv* and return the maximum value. We may also produce the pair with this result and the initial list of adventurers.

   ```
   addCrossTime :: [Adventurer] -> (Int, [Adventurer])
   addCrossTime = split (maximum . (map getTimeAdv)) id
   ```

4. add the lantern to the group that is going to cross — they need the lantern to cross.

   ```
   addLantern :: (Int, [Adventurer]) -> (Int, [Either Adventurer ()])
   addLantern = id >< ((Right () :) . map Left)
   ```

   This returns the list of objects that are going to cross and the time needed to do it.

5. finally, recurr to the function map *mChangeState* to change the state of the elements which are going to cross, i.e., compute our possible moves and encapsulate it in the monad using the composition $LSD \cdot \text{map}\ Duration$. Yes, we are missing something — how to produce the path (or the trace)! For now, let us just appreciate the final function. How we get the path will be explained after.

   ```
   allValidPlays :: State -> AdventurersLog State
   allValidPlays s = ALog $ map Duration $
       map (id >< (split (toTrace s) id) . (mCS s)) t where
     t = (map (addLantern.addCrossTime) . comUpTo2 . advsWhereLanternIs) s
     mCS = flip mChangeState
     toTrace s s' = printTrace (state2List s, state2List s')
   ```

The trace log As we saw, our monad *AdventurersLog* keeps the trace by calling the function $toTrace :: State \to State \to String$. But what does this function do? First, we can see that, according to the representation of the state, adventurers can be represented by indexes. We take advantage of this to be able to present an elegant trace of the moves. For example, if the previous state is $[False, False, False, False, False]$ and the current state is $[True, True, False, False, True]$, we know that $P1$ and $P2$ have crossed (because the first two and the last elements and diferent). So, we can simply compare element to element and, if they are different, we keep the index. In the previous example, it would return $[0, 1, 4]$ — index 4 represents the lantern, and because we assume that the movements are always valid, we can ignore that.

```
index2Adv :: Int -> String
index2Adv 0 = "P1"
index2Adv 1 = "P2"
index2Adv 2 = "P5"
index2Adv 3 = "P10"

indexesWithDifferentValues :: Eq a => ([a], [a]) -> [Int]
indexesWithDifferentValues (l1, l2) = aux l1 l2 0 where
  aux :: Eq a => [a] -> [a] -> Int -> [Int]
  aux [] l _ = []
  aux l [] _ = []
  aux (h1:t1) (h2:t2) index = if h1 /= h2 then index : aux t1 t2 (index + 1)
                              else aux t1 t2 (index + 1)
```

The result $[0, 1, 4]$ means that "$P1$ and $P2$ crosses". We now have to automate this (pretty) print. We only need to ignore the lantern index (4), convert the indexes to the respective adventurers and define a print function for them.

```
printTrace :: ([Bool], [Bool]) -> String
printTrace = prettyLog . (map index2Adv) . init . indexesWithDifferentValues

prettyLog :: [String] -> String
prettyLog = Cp.cond ((>1) . length) f ((++ " cross\n") . head) where
    f = (++" crosses\n") . conc . ((concat . map (++" and ")) >< id) .
    (split init last)
```

Back to function *allValidPlays*, we do, for a given state s and each following state s',

```
toTrace :: State -> State -> String
toTrace = curry $ printTrace . (state2List >< state2List)
```

So, this representation is done right in the calculation of the possible moves. At the end, we just need to get that already prepared trace. Now, we shall see the trace of the optimal play which shows how elegant the log is.

Solving the problem We may define a function that, for a given number n and an initial state, calculates all possible n-sequences of moves that the adventures can make. For that, we may took advantage of the **do** notation — let the monad do the work!

```
exec :: Int -> State -> AdventurersLog State
exec 0 s = allValidPlays s
exec n s = do ps <- exec (n-1) s
              allValidPlays ps
```

The previous functions is good, but not so good — we don't know how many sequences are needed to reach the end state. It would be much better if we could execute all possible sequences of moves that the adventures can make for a given state untill it fulfills a predicate over a state (passed as a parameter). Additionally, it also returns the number of moves needed to fulfill that predicate.

```
execPred :: (State -> Bool) -> State -> (Int, AdventurersLog State)
execPred p s = aux p s 0 where
    aux p s it =
        let st = exec it s
            res = filter pred (map remDur (remALog st)) in
                if length (res) > 0 then ((it+1) , ALog (map Duration res))
                else aux p s (it+1) where
                    remDur (Duration a) = a
                    pred (_, (_,s)) = p s
```

We may use this last function to solve the problem and see who is right. For that, we define one more function to see if it is possible for all adventurers to be on the other side in $\leqslant n$ minutes and how many moves are needed for that.

```
leqX :: Int -> (Int, Bool)
leqX n = if res then (it, res)
                else (-1, res) where
                  res = length (filter p (map remDur (remALog l))) > 0
                  (it,l) = execPred (== cEnd) cInit
                  p (d,(_,_)) = d <= n
                  remDur (Duration a) = a
```

So let us see who was right!

```
> p2 (leqX 17) && p1 (leqX 17) <= 5
True
```

So, it is possible for all adventurers to be on the other side in $\leqslant 17$ minutes and not exceeding 5 moves. Actually, since `p2 (leqX 16)` is *False*, 17 minutes is the the optimal time for solving the problem (with exactly 5 moves). One could also get that information by executing the following function *optimalTrace*, to which the reader is invited to analyze.

```
optimalTrace :: IO ()
optimalTrace =
    putStrLn . t . map remDur . remALog . p2 $ execPred (== cEnd) cInit where
    t = prt . (split (head . map p1) (map (p1.p2))) . pairFilter .
        split (minimum . map p1) id
    remDur (Duration a) = a
    pairFilter (d, l) = filter (\ (d', (_, _)) -> d == d') l
```

```
    p = Cp.cond ((>1) . length) p' head
    p' = conc . split (concat . map ((++("\nOR\n\n"))) . init) last
    prt (d, l) = (p l) ++ "\nin " ++ (show d) ++ " minutes."
```

Result:

```
> optimalTrace
P1 and P2 crosses
P1 cross
P5 and P10 crosses
P2 cross
P1 and P2 crosses

OR

P1 and P2 crosses
P2 cross
P5 and P10 crosses
P1 cross
P1 and P2 crosses

in 17 minutes.
```

Bibliography

R. Neves. Cyber-physical programming course. https://haslab.github.io/MFP/PCF/2122/index.

J.N. Oliveira. Aulas teóricas de cálculo de programas. https://www4.di.uminho.pt/~jno/media/cp/.

J.N. Oliveira. Program Design by Calculation, 2022. Draft of textbook in preparation, current version: September 2022. Informatics Department, University of Minho (PDF).