Chapter 1

An Introduction to Pointfree Programming

Everybody is familiar with the concept of a *function* since the school desk. The functional intuition traverses mathematics from end to end because it has a solid semantics rooted on a well-known mathematical structure — the class "all" sets and set-theoretical functions.

Functional programming literally means "programming with functions". Programming languages such as LISP, or HASKELL allow us to program with functions. However, the functional intuition is far more reaching than producing code which runs on a computer. Since the pioneering work of John McCarthy — the inventor of LISP — in the early 1960s, one knows that other branches of programming can be structured, or expressed functionally. The idea of producing programs by *calculation*, that is to say, that of calculating efficient programs out of abstract, inefficient ones has a long tradition in functional programming.

This book is structured around the idea that functional programming can be used as a basis for teaching programming as a whole, from the successor function $n \mapsto n + 1$ to large information system design.

This chapter provides a light-weight introduction to the theory of functional programming. Its emphasis is on explaining how to construct new functions out of other functions using a minimal set of predefined functional combinators. This leads to a programming style which is *point free* in the sense that function descriptions dispense with variables (definition *points*).

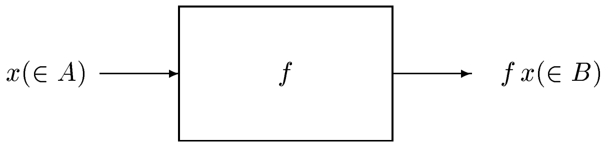
Many technical issues are deliberately ignored and deferred to later chapters. Most programming examples will be provided in the HASKELL functional programming language. Appendix A includes the listings of some HASKELL modules which complement the HUGS *Standard Prelude* (which is based very closely on the *Standard Prelude* for HASKELL 1.4.) and help to "animate" the main concepts introduced in this chapter.

1.1 Introducing functions and types

The definition of a function

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

can be regarded as a kind of "process": it is a "black box" which produces an output once it is supplied with an input:



From another viewpoint, f can be regarded as a kind of "contract": it commits itself to producing a B-value provided it is supplied with an A-value. How is such a value produced? In many situations one wishes simply to ignore it. In others, one may want to inspect the internals of the "black box" only to know the function's *computation rule*. For instance,

$$\begin{array}{rcl} \mathit{succ} & : & \mathbb{N} \longrightarrow \mathbb{N} \\ \mathit{succ}\ n & \stackrel{\mathrm{def}}{=} & n+1 \end{array}$$

expresses the computation rule of the *successor* function — the function *succ* which finds "the next natural number" — in terms of natural number addition and of natural number 1. What we above meant by a "contract" corresponds to the *signature* of the function, which is expressed by arrow $\mathbb{N} \longrightarrow \mathbb{N}$ and which, by the way, can be fulfilled by other functions, *e.g.* $\mathit{sq}\ n \stackrel{\mathrm{def}}{=} n^2$.

In programming terminology one says that *succ* and *sq* have the same "type". Types play a prominent rôle in functional programming (as they do in other programming paradigms). Informally, they provide the "glue", or interfacing material, for putting functions together to obtain more complex functions. Formally, a "type checking" discipline can be expressed in terms of compositional rules which check for functional expression wellformedness.

It has become standard to use arrows to denote function signatures or function types, recall (1.1). In this book the following variants will be used interchangeably to denote the fact that function f accepts arguments of type A and produces results of type B: $f : B \longleftarrow A$, $f : A \longrightarrow B$, $B \xleftarrow{f} A$ or $A \xrightarrow{f} B$. This corresponds to writing `f :: a -> b` in the HASKELL functional programming language, where type variables are denoted by lowercase letters. A will be referred to as the *domain* of f and B will be referred to as the *codomain* of f. Both A and B are symbols which denote sets of values, very often called *types*.

1.2 Functional application

What do we want functions for? If we ask this question to a physician or engineer the answer is very likely to be: one wants functions for modelling and reasoning about the behaviour of real things.

For instance, function $\mathit{distance}\, t = 60 \times t$ could be written by a school physics student to model the distance (in, say, kilometers) a car will drive (per hour) at average speed $60km/hour$. When questioned about how far the car has gone in 2.5 hours, such a model provides an immediate answer: just evaluate $\mathit{distance}\ 2.5$ to obtain $150km$.

So one gets one of the purposes of functions: we want them to be *applied* to arguments in order to obtain results. Functional *application* is denoted by juxtaposition, *e.g.* $f\,a$ for $B \xleftarrow{f} A$ and $a \in A$, and associates to the left: $f\,x\,y$ denotes $(f\,x)\,y$ rather than $f\,(x\,y)$.

1.3 Functional equality and composition

Application is not everything we want to do with functions. Very soon our physics student will be able to talk about properties of the *distance* model, for instance that property

$$\mathit{distance}\,(2 \times t) = 2 \times (\mathit{distance}\, t) \tag{1.2}$$

holds. Later on, we could learn from her or him that the same property can be restated as $\mathit{distance}\,(\mathit{twice}\, t) = \mathit{twice}\,(\mathit{distance}\, t)$, by introducing function $\mathit{twice}\, x \stackrel{\mathrm{def}}{=} 2 \times x$. Or even simply as

$$\mathit{distance} \cdot \mathit{twice} = \mathit{twice} \cdot \mathit{distance} \tag{1.3}$$

where "·" denotes function-arrow chaining, as suggested by drawing

$$\begin{tikzcd} \mathbb{R} \arrow[d, "\mathit{distance}"'] & \mathbb{R} \arrow[l, "\mathit{twice}"'] \arrow[d, "\mathit{distance}"] \\ \mathbb{R} & \mathbb{R} \arrow[l, "\mathit{twice}"] \end{tikzcd} \tag{1.4}$$

where both space and time are modelled by real numbers.

This trivial example illustrates some relevant facets of the functional programming trend. Which version of the property presented above is "better"? the version explicitly mentioning variable t and requiring parentheses (1.2)? the version hiding variable t but resorting to function $twice$ (1.3)? or drawing (1.4)?

Expression (1.3) is clearly more compact than (1.2). The trend for notation economy and compactness is well-known throughout the history of mathematics. In the 16th century, for instance, algebrists would write *12.cu.p̃.18.ce.p̃.27.co.p̃.17* for what is nowadays written as $12x^3+18x^2+27x+17$. We may find such *syncopated* notation odd, but should not forget that at its time it was replacing even more obscure expression denotations.

Why do people look for compact notations? A compact notation leads to shorter documents (less lines of code in programming) in which patterns are easier to identify and to reason about. Properties can be stated in clear-cut, one-line long equations which are easy to memorize. And diagrams such as (1.4) can be easily drawn which enable us to visualize maths in a graphical format.

Some people will argue that such compact "pointfree" notation (that is, the notation which hides variables, or function "definition points") is too cryptic to be useful as a practical programming medium. In fact, pointfree programming languages such as Iverson's APL or Backus' FP have been more respected than loved by the programmers community. Virtually all commercial programming languages require variables and so implement the more traditional "pointwise" notation.

Throughout this book we will adopt both, depending upon the context. Our chosen programming medium — HASKELL — blends the pointwise and pointfree programming styles both in a quite successful way. In order to switch from one to the other, we need two "bridges": one lifting equality to the functional level and the other lifting application.

Concerning equality, note that the "$=$" sign in (1.2) differs from that in (1.3): while the former states that two real numbers are the same number, the latter states that two $\mathbb{R} \longrightarrow \mathbb{R}$ functions are the same function. Formally, we will say that two functions $f, g : A \longrightarrow B$ are equal if they agree at pointwise-level, that is, if

$$f = g \quad \textit{iff} \quad \forall a \in A : f\,a \;=_B\; g\,a \tag{1.5}$$

holds, where $=_B$ denotes equality at B-level.

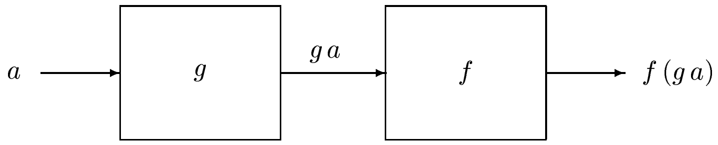
Concerning application, the pointfree style replaces it by the more generic concept of functional *composition* suggested by function-arrow chaining: wherever two functions are such that the target type of one of them, say $B \xleftarrow{g} A$ is the same as the source type of the other, say $C \xleftarrow{f} B$, then another function can be defined, $C \xleftarrow{f \cdot g} A$ — called the *composition* of f and g, or "f after g" — which "glues" f and g together:

$$(f \cdot g)\,a \quad \stackrel{\rm def}{=} \quad f\,(g\,a) \tag{1.6}$$

This situation is pictured by the following arrow-diagram

$$\begin{tikzcd} B \arrow[d, "f"'] & A \arrow[l, "g"'] \arrow[dl, "f \cdot g"] \\ C \end{tikzcd} \tag{1.7}$$

or by block-diagram



Therefore, the type-rule associated to functional composition can be expressed as follows:

$$\frac{\begin{array}{c} B \xleftarrow{f} C \\ C \xleftarrow{g} A \end{array}}{B \xleftarrow{f \cdot g} A}$$

Composition is certainly the most basic of all functional combinators. It is the first kind of "glue" which comes to mind when programmers need to combine, or chain functions (or processes) to obtain more elaborate functions (or processes) [1]. This is because of one of its most relevant properties,

$$(f \cdot g) \cdot h \quad = \quad f \cdot (g \cdot h) \tag{1.8}$$

which shares the pattern of, for instance

$$(a + b) + c \quad = \quad a + (b + c)$$

and so is called the *associative* property of composition. This enables us to move parentheses around in pointfree expressions involving functional compositions, or even to omit them, for instance by writing $f \cdot g \cdot h \cdot i$ as an abbreviation of $((f \cdot g) \cdot h) \cdot i$, or of $(f \cdot (g \cdot h)) \cdot i$, or of $f \cdot ((g \cdot h) \cdot i)$, *etc.* For a chain of n-many function compositions the notation $\bigcirc_{i=1}^{n} f_i$ will also be acceptable.

1.4 Identity functions

How free are we to fulfill the "give me an A and I will give you a B" contract of equation (1.1)? In general, the choice of f is not unique. Some fs will do as little as possible while others will laboriously compute non-trivial outputs. At one of the extremes, we find functions which "do nothing" for us, that is, the added-value of their output when compared to their input amounts to nothing:

$$f\,a = a$$

[1] It even has a place in script languages such as UNIX's, where `f | g` is the shell counterpart of $g \cdot f$, for appropriate "processes" f and g.

In this case $B = A$, of course, and f is said to be the *identity* function on A:

$$\begin{array}{rcl} id_A & : & A \longleftarrow A \\ id_A\, a & \stackrel{\mathrm{def}}{=} & a \end{array} \tag{1.9}$$

Note that every type X "has" its identity id_X. Subscripts will be omitted wherever implicit in the context. For instance, the arrow notation $\mathbb{N} \xleftarrow{id} \mathbb{N}$ saves us from writing $id_{\mathbb{N}}$, *etc.*. So, we will often refer to "the" identity function rather than to "an" identity function.

How useful are identity functions? At first sight, they look fairly uninteresting. But the interplay between composition and identity, captured by the following equation,

$$f \cdot id = id \cdot f = f \tag{1.10}$$

will be appreciated later on. This property shares the pattern of, for instance,

$$a + 0 = 0 + a = a$$

This is why we say that id is the *unit* of composition. In a diagram, (1.10) looks like this:

$$\begin{tikzcd} A \arrow[d, "f"'] & A \arrow[l, "id"'] \arrow[d, "f"] \\ B & B \arrow[l, "id"] \end{tikzcd} \tag{1.11}$$

Note the graphical analogy of diagrams (1.4) and (1.11). Diagrams of this kind are very common and express important properties of functions, as we shall very often see.

1.5 Constant functions

Opposite to the identity functions, which do not lose any information, we find functions which lose all (or almost all) information. Regardless of their input, the output of these functions is always the same value.

Let C be a nonempty data domain and let and $c \in C$. Then we define the *everywhere c* function as follows, for arbitrary A:

$$\begin{array}{rcl} \underline{c} & : & A \longrightarrow C \\ \underline{c}\, a & \stackrel{\mathrm{def}}{=} & c \end{array} \tag{1.12}$$

The following property defines constant functions at pointfree level,

$$\underline{c} \cdot f \quad = \quad \underline{c} \tag{1.13}$$

and is depicted by a diagram similar to (1.11):

$$\begin{tikzcd} C & A \arrow[l, "\underline{c}"'] \arrow[d, "f"] \\ C \arrow[from=u, "id"'] & B \arrow[l, "\underline{c}"] \end{tikzcd} \tag{1.14}$$

Note that, strictly speaking, symbol $\underline{c}$ denotes two different functions in diagram (1.14): one, which we should have written $\underline{c}_A$, accepts inputs from A while the other, which we should have written $\underline{c}_B$, accepts inputs from B. As with identity functions, subscripts will be omitted wherever implicit in the context.

Exercise 1.1 The HUGS *Standard Prelude* provides for constant functions: you write `const c` for $\underline{c}$. Check that HUGS assigns the same type to expressions `f . const c` and `const (f c)`, for every `f` and `c`. What else can you say about these functional expressions? Justify.
□

1.6 Monics and epics

Identity functions and constant functions are the limit points of the functional spectrum with respect to information preservation. All the other functions are in between: they lose "some" information, which is regarded as uninteresting for some reason. This remark supports the following aphorism about a facet of functional programming: it is the *art* of transforming or losing information in a controlled and precise way. That is to say, the art of constructing the exact observation of data which fits in a particular context or requirement.

How do functions lose information? Basically in two different ways: they may be "blind" enough to confuse different inputs, by mapping them onto the same output, or they may ignore values of their codomain. For instance, $\underline{c}$ confuses *all* inputs by mapping them all onto c. Moreover, it ignores all values of its codomain apart from c.

Functions which do not confuse inputs are called *monics* (or injective functions) and obey the following property: $B \xleftarrow{f} A$ is *monic* if, for every pair of functions $A \xleftarrow{h,k} C$, if $f \cdot h = f \cdot k$ then $h = k$, *cf.* diagram

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

(f is "cancellable on the left").

It is easy to check that "the" identity function is monic,

$$\begin{array}{rl} & id \cdot h = id \cdot k \Rightarrow h = k \\ \leftrightarrow & \quad \{ \text{ by } (1.10) \} \end{array}$$

$$
\begin{array}{cl}
& h = k \Rightarrow h = k \\
\leftrightarrow & \qquad \{ \text{ predicate logic} \} \\
& \mathrm{TRUE}
\end{array}
$$

and that any constant function $\underline{c}$ is not monic:

$$
\begin{array}{cl}
& \underline{c} \cdot h = \underline{c} \cdot k \Rightarrow h = k \\
\leftrightarrow & \qquad \{ \text{ by (1.13)} \} \\
& \underline{c} = \underline{c} \Rightarrow h = k \\
\leftrightarrow & \qquad \{ \text{ function equality is reflexive} \} \\
& \mathrm{TRUE} \Rightarrow h = k \\
\leftrightarrow & \qquad \{ \text{ predicate logic} \} \\
& h = k
\end{array}
$$

So the implication does not hold in general (only if $h = k$).

Functions which do not ignore values of their codomain are called *epics* (or surjective functions) and obey the following property: $A \xleftarrow{f} B$ is *epic* if, for every pair of functions $C \xleftarrow{h,k} A$, if $h \cdot f = k \cdot f$ then $h = k$, *cf.* diagram

$$
\begin{tikzcd}
C & A \arrow[l, shift right, "k"'] \arrow[l, shift left, "h"] & B \arrow[l, "f"]
\end{tikzcd}
$$

(f is "cancellable on the right").

As expected, identity functions are epic:

$$
\begin{array}{cl}
& h \cdot id = k \cdot id \Rightarrow h = k \\
\leftrightarrow & \qquad \{ \text{ by (1.10)} \} \\
& h = k \Rightarrow h = k \\
\leftrightarrow & \qquad \{ \text{ predicate logic} \} \\
& \mathrm{TRUE}
\end{array}
$$

Exercise 1.2 Under what circumstances is a constant function epic? Justify.
□

1.7 Isos

A function $B \xleftarrow{f} A$ which is both monic and epic is said to be *iso* (an isomorphism, or a bijective function). In this situation, f always has an *inverse* $B \xrightarrow{f^{-1}} A$, which is such that

$$f \cdot f^{-1} = id_B \quad \wedge \quad f^{-1} \cdot f = id_A \tag{1.15}$$

(*i.e.* f is *invertible*).

Isomorphisms are very important functions because they convert data from one "format", say A, to another format, say B, without losing information. So f and and f^{-1} are faithful protocols between the two formats A and B. Of course, these formats contain the same "amount" of information, although the same data adopts a different "shape" in each of them. In mathematics, one says that A is *isomorphic* to B and one writes $A \cong B$ to express this fact.

Isomorphic data domains are regarded as "abstractly" the same. Note that, in general, there is a wide range of isos between two isomorphic data domains. For instance, let $\mathsf{Weekday}$ be the set of weekdays,

$$\begin{aligned}&\mathsf{Weekday} = \\ &\quad \{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday\}\end{aligned}$$

and let symbol $\mathsf{7}$ denote the set $\{1, 2, 3, 4, 5, 6, 7\}$, which is the *initial segment* of $\mathbb{N}$ containing exactly seven elements. The following function f, which associates each weekday with its "ordinal" number,

$$\begin{aligned}&f \;:\; \mathsf{Weekday} \longrightarrow \mathsf{7}\\ &f\, Monday = 1\\ &f\, Tuesday = 2\\ &f\, Wednesday = 3\\ &f\, Thursday = 4\\ &f\, Friday = 5\\ &f\, Saturday = 6\\ &f\, Sunday = 7\end{aligned}$$

is iso (guess f^{-1}). Clearly, $f\, d = i$ means "d is the i-th day of the week". But note that function $g\, d \stackrel{\text{def}}{=} rem(f\, d, 7) + 1$ is also an iso between $\mathsf{Weekday}$ and $\mathsf{7}$. While f regards $Monday$ the first day of the week, g places $Sunday$ in that position. Both f and g are witnesses of isomorphism

$$\mathsf{Weekday} \quad \cong \quad \mathsf{7} \tag{1.16}$$

Note that all classes of functions referred to so far — constants, identities, epics, monics and isos — are closed under composition, that is, the composition of two constants is a constant, the composition of two epics is epic, *etc.*

1.8 Gluing functions which do not compose — products

Function composition has been presented above as the basis for gluing functions together to build more complex functions. However, not every two functions can be glued together by composition. For instance, functions $f : A \longleftarrow C$ and $g : B \longleftarrow C$ do not compose with each other because the domain of one of them is not the codomain of the other. However, both f and g share the same domain C. So, something we can do about gluing f and g is to draw a diagram expressing this fact, something like

$$\begin{tikzcd} A & & B \\ & C \arrow[ul, "f"] \arrow[ur, "g"'] & \end{tikzcd}$$

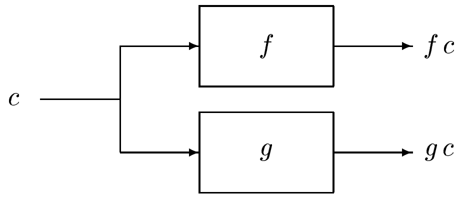
Because f and g share the same domain, their outputs can be paired, that is, we may write ordered pair $(f\,c, g\,c)$ for each $c \in C$. Such pairs belong to the Cartesian product of A and B, that is, to the set

$$A \times B \stackrel{\rm def}{=} \{(a, b) \mid a \in A \wedge b \in B\}$$

So we may think of the operation which pairs the outputs of f and g as a new function combinator $\langle f, g\rangle$ defined as follows:

$$\begin{array}{rcl} \langle f, g\rangle & : & C \longrightarrow A \times B \\ \langle f, g\rangle\, c & \stackrel{\rm def}{=} & (f\,c, g\,c) \end{array} \tag{1.17}$$

Function combinator $\langle f, g\rangle$ is pronounced "f *split* g" (or "pair f and g") and can be depicted by the following "block", or "data flow" diagram:



$\langle f, g\rangle$ keeps the information of both f and g in the same way Cartesian product $A \times B$ keeps the information of A and B. So, in the same way A data or B data can be retrieved from $A \times B$ data via the implicit *projections* π_1 or π_2,

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B \tag{1.18}$$

defined by

$$\pi_1(a, b) = a \quad \text{and} \quad \pi_2(a, b) = b$$

can f and g be retrieved from $\langle f, g\rangle$ via the same projections:

$$\pi_1 \cdot \langle f, g\rangle = f \quad \text{and} \quad \pi_2 \cdot \langle f, g\rangle = g \tag{1.19}$$

This fact (or pair of facts) will be referred to as the $\times$-*cancellation* property and is illustrated in the following diagram which puts things together:

$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\ & C \arrow[ul, "f"] \arrow[u, "\langle f,g\rangle"'] \arrow[ur, "g"'] & \end{tikzcd} \tag{1.20}$$

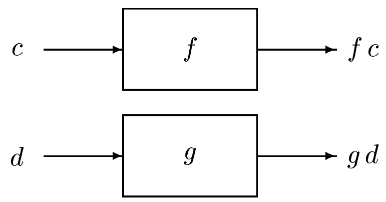
In summary, the type-rule associated to the "split" combinator is expressed by

$$\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} C \\ \hline A \times B \xleftarrow{\langle f,g\rangle} C \end{array}$$

A *split* arises wherever two functions do not compose but share the same domain. What about gluing two functions which fail such a requisite, *e.g.*

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

The $\langle f, g\rangle$ *split* combination does not work any more. But a way to "unify" the domains of f and g, C and D respectively, is to regard them as targets of the projections π_1 and π_2 of $C \times D$. That is to say, expression $\langle f \cdot \pi_1, g \cdot \pi_2\rangle$ is well-typed, having domain $C \times D$ and codomain $A \times B$. It corresponds to the "parallel" application of f and g which is suggested by the following data-flow diagram:



Functional combination $\langle f \cdot \pi_1, g \cdot \pi_2\rangle$ appears very often and deserves special notation — it will be expressed by $f \times g$. So, by definition, we have

$$f \times g \stackrel{\text{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2\rangle \tag{1.21}$$

which is pronounced "product of f and g" and has typing-rule

$$\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \\ \hline A \times B \xleftarrow{f \times g} C \times D \end{array} \tag{1.22}$$

Note the overloading of symbol "$\times$", which is used to denote both Cartesian product and functional product. The adoption of this notation will be fully justified later on.

What is the interplay among functional combinators $f \cdot g$ (composition), $\langle f, g\rangle$ (*split*) and $f \times g$ (product) ? Composition and *split* relate to each other via the following property, known as $\times$-*fusion*:

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \arrow[ul, "g"] \arrow[u, "\langle g,h\rangle"'] \arrow[ur, "h"'] & \\
& D \arrow[u, "f"] \arrow[uul, bend left, "g \cdot f"] \arrow[uur, bend right, "h \cdot f"'] &
\end{tikzcd}
\qquad \langle g, h\rangle \cdot f = \langle g \cdot f, h \cdot f\rangle \tag{1.23}$$

This shows that *split* is right-distributive with respect to composition. Left-distributivity does not hold but there is something we can say about $f \cdot \langle g, h\rangle$ in case $f = i \times j$:

$$\begin{array}{cl}
& (i \times j) \cdot \langle g, h\rangle \\
= & \qquad \{ \text{ by (1.21)} \} \\
& \langle i \cdot \pi_1, j \cdot \pi_2\rangle \cdot \langle g, h\rangle \\
= & \qquad \{ \text{ by } \times\text{-fusion (1.23)} \} \\
& \langle (i \cdot \pi_1) \cdot \langle g, h\rangle, (j \cdot \pi_2) \cdot \langle g, h\rangle\rangle \\
= & \qquad \{ \text{ by (1.8)} \} \\
& \langle i \cdot (\pi_1 \cdot \langle g, h\rangle), j \cdot (\pi_2 \cdot \langle g, h\rangle)\rangle \\
= & \qquad \{ \text{ by } \times\text{-cancellation (1.19)} \} \\
& \langle i \cdot g, j \cdot h\rangle
\end{array}$$

The law we have just derived is known as $\times$-*absorption*. (The intuition behind this terminology is that "*split* absorbs $\times$" as a special kind of fusion.) It is a consequence of $\times$-fusion and $\times$-cancellation and is depicted as follows:

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
D \arrow[u, "i"] & D \times E \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[u, "i \times j"] & E \arrow[u, "j"'] \\
& C \arrow[ul, "g"] \arrow[u, "\langle g,h\rangle"'] \arrow[ur, "h"'] &
\end{tikzcd}
\qquad (i \times j) \cdot \langle g, h\rangle = \langle i \cdot g, j \cdot h\rangle \tag{1.24}$$

This diagram provides us with two further results about products and projections which can be easily justified:

$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{1.25}$$
$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{1.26}$$

Two special properties of $f \times g$ are presented next. The first one expresses a kind of "bi-distribution" of $\times$ with respect to composition:

$$(g \cdot h) \times (i \cdot j) \quad = \quad (g \times i) \cdot (h \times j) \tag{1.27}$$

We will refer to this property as the $\times$*-functor property*. The other property, which we will refer to as the $\times$*-functor-id property*, has to do with identity functions:

$$id_A \times id_B \quad = \quad id_{A \times B} \tag{1.28}$$

These two properties will be identified as the *functorial properties* of product. This choice of terminology will be explained later on.

Let us finally analyse the particular situation in which a *split* is built involving projections π_1 and π_2 only. These exhibit interesting properties, for instance $\langle \pi_1, \pi_2 \rangle = id$. This property is known as $\times$*-reflexion* and is depicted as follows:

$$\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& A \times B \arrow[ul, "\pi_1"] \arrow[u, "id_{A \times B}"] \arrow[ur, "\pi_2"'] &
\end{tikzcd} \qquad \langle \pi_1, \pi_2 \rangle = id_{A \times B} \tag{1.29}$$

What about $\langle \pi_2, \pi_1 \rangle$? This corresponds to a diagram

$$\begin{tikzcd}
B & B \times A \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & A \\
& A \times B \arrow[ul, "\pi_2"] \arrow[u, "\langle \pi_2, \pi_1 \rangle"] \arrow[ur, "\pi_1"'] &
\end{tikzcd}$$

which looks very much the same if submitted to a 180^o clockwise rotation (A and B swap with each other). This suggests that $swap$ (the name we adopt for $\langle \pi_2, \pi_1 \rangle$) is its own inverse, as can be checked easily as follows:

$$\begin{array}{cl}
& swap \cdot swap \\
= & \qquad \{ \text{ by definition } swap \stackrel{\text{def}}{=} \langle \pi_2, \pi_1 \rangle \} \\
& \langle \pi_2, \pi_1 \rangle \cdot swap \\
= & \qquad \{ \text{ by } \times\text{-fusion (1.23)} \} \\
& \langle \pi_2 \cdot swap, \pi_1 \cdot swap \rangle \\
= & \qquad \{ \text{ by definition of } swap \} \\
& \langle \pi_2 \cdot \langle \pi_2, \pi_1 \rangle, \pi_1 \cdot \langle \pi_2, \pi_1 \rangle \rangle \\
= & \qquad \{ \text{ by } \times\text{-cancellation (1.19)} \} \\
& \langle \pi_1, \pi_2 \rangle \\
= & \qquad \{ \text{ by } \times\text{-reflexion (1.29)} \} \\
& id
\end{array}$$

Therefore, *swap* is iso and establishes the following isomorphism

$$A \times B \quad \cong \quad B \times A \tag{1.30}$$

which is known as the *commutative property* of product.

The "product datatype" $A \times B$ is essential to information processing and is available in virtually every programming language. In HASKELL one writes `(A,B)` to denote $A \times B$, for A and B two predefined datatypes, `fst` to denote π_1 and `snd` to denote π_2. In the C programming language this datatype is called the "struct datatype",

```
struct {
   A first;
   B second;
};
```

while in PASCAL it is called the "record datatype":

```
record
   first:  A;
   second:  B
end;
```

Isomorphism (1.30) can be re-interpreted in this context as a guarantee that *one does not lose (or gain) anything in swapping fields in record datatypes.* C or PASCAL programmers know also that record-field nesting has the same status, that is to say that, for instance, datatype

```
record
    F: A;
    S: record
          F: B;
          S: C;
       end
end;
```

is abstractly the same as

```
record
    F: record
          F: A;
          S: B
       end;
    S: C;
end;
```

In fact, this is another well-known isomorphism, known as the *associative property* of product:

$$A \times (B \times C) \quad \cong \quad (A \times B) \times C \tag{1.31}$$

This is established by $A \times (B \times C) \xleftarrow{assocr} (A \times B) \times C$, which is pronounced "associate to the right" and is defined by

$$assocr \stackrel{\text{def}}{=} \langle \pi_1 \cdot \pi_1, \langle \pi_2 \cdot \pi_1, \pi_2 \rangle \rangle \tag{1.32}$$

Section A.1 in the appendix lists an extension to the HUGS *Standard Prelude*, called `Set.hs`, which makes isomorphisms such as *swap* and *assocr* available. In this module, the concrete syntax chosen for $\langle f, g \rangle$ is `split f g` and the one chosen for $f \times g$ is `f >< g`.

Exercise 1.3 Show that $assocr$ is iso by conjecturing its inverse $assocl$ and proving that functional equality $assocr \cdot assocl = id$ holds.
□

Exercise 1.4 Use (1.21) to prove properties (1.27) and (1.28).
□

1.9 Gluing functions which do not compose — coproducts

The *split* functional combinator arose in the previous section as a kind of glue for combining two functions which do not compose but share the same domain. The "dual" situation of two non-composable functions $f : C \longleftarrow A$ and $g : C \longleftarrow B$ which however share the same codomain is depicted in

$$\begin{tikzcd} A \arrow[dr, "f"'] & & B \arrow[dl, "g"] \\ & C & \end{tikzcd}$$

It is clear that the kind of glue we need in this case should be able to apply f in case we are on the "A-side" or to apply g in case we are on the "B-side". Let us write $[\, f, g \,]$ to denote the new kind of combinator. Its codomain will be C. What about its domain?

We need to describe the datatype which is "either an A or a B". Since A and B are sets, we may think of $A \cup B$ as such a datatype. This works in case A and B are disjoint sets, but wherever the intersection $A \cap B$ is non-empty it is undecidable whether a value $x \in A \cap B$ is an "A-value" or a "B-value". In the limit, if $A = B$ then $A \cup B = A = B$, that is to say, we have not invented a new datatype at all. These difficulties can be circumvented by resorting to disjoint union:

$$A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$$

The values of $A + B$ can be thought of as "copies" of A or B values which are "stamped" with different tags in order to guarantee that values which are simultaneously in A and B do not get mixed up. The tagging functions i_1 and i_2 are called *injections*:

$$i_1\, a = (t_1, a) \quad , \quad i_2\, b = (t_2, b) \tag{1.33}$$

Knowing the exact values of tags t_1 and t_2 is not essential to understanding the concept of a disjoint union. It suffices to know that i_1 and i_2 tag differently and consistently.

For instance, the following realizations of $A + B$ in the C programming language,

```
struct {
   int tag; /* 1,2 */
   union {
      A ifA;
      B ifB;
   } data;
};
```

or in PASCAL,

```
record
   case
    tag:  integer
        of x =
     1:  (P:A);
     2:  (S:B)
end;
```

adopt integer tags. In the HUGS *Standard Prelude*, which is based very closely on the *Standard Prelude* for HASKELL 1.4., the $A + B$ datatype is realized by

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

So, `Left` and `Right` can be thought of as the injections i_1 and i_2 in this realization.

At this level of abstraction, disjoint union $A + B$ is called the *coproduct* of A and B, on top of which we define the new combinator $[\, f, g\,]$ (pronounced "either f or g") as follows:

$$\begin{array}{rcl} [\, f,g\,] & : & A + B \longrightarrow C \\ [\, f,g\,]\, x & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{lcl} x = i_1\, a & \Rightarrow & f\, a \\ x = i_2\, b & \Rightarrow & g\, b \end{array} \right. \end{array} \tag{1.34}$$

As we did for products, we can express all this in a single diagram:

$$\begin{tikzcd} A \arrow[r, "i_1"] \arrow[dr, "f"'] & A + B \arrow[d, "{[\, f,g\,]}"] & B \arrow[l, "i_2"'] \arrow[dl, "g"] \\ & C & \end{tikzcd} \tag{1.35}$$

It is interesting to note how similar this diagram is to the one drawn for products — one just has to reverse the arrows, replace projections by injections and the *split* arrow by the *either* one. This expresses the fact that *product* and *coproduct* are *dual* mathematical constructs (compare with *sine* and *cosine*). This duality is of great conceptual economy because everything we can say about product $A \times B$ can be rephrased to coproduct $A + B$. For instance, we may introduce the sum of two functions $f + g$ as the notion dual to product $f \times g$:

$$f + g \stackrel{\mathrm{def}}{=} [\, i_1 \cdot f, i_2 \cdot g\,] \tag{1.36}$$

The following list of $+$-laws provides eloquent evidence of this duality:

$+$-cancellation :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "g"'] & A+B \arrow[d, "{[\,g,h\,]}"] & B \arrow[l, "i_2"'] \arrow[dl, "h"] \\
& C &
\end{tikzcd}
\qquad [\,g,h\,] \cdot i_1 = g \ , \ [\,g,h\,] \cdot i_2 = h \tag{1.37}$$

$+$-reflexion :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "i_1"'] & A+B \arrow[d, "id_{A+B}"] & B \arrow[l, "i_2"'] \arrow[dl, "i_2"] \\
& A+B &
\end{tikzcd}
\qquad [\,i_1, i_2\,] = id_{A+B} \tag{1.38}$$

$+$-fusion :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "g"'] \arrow[ddr, "f \cdot g"'] & A+B \arrow[d, "{[\,g,h\,]}"] & B \arrow[l, "i_2"'] \arrow[dl, "h"] \arrow[ddl, "f \cdot h"] \\
& C \arrow[d, "f"] & \\
& D &
\end{tikzcd}
\qquad f \cdot [\,g,h\,] = [\,f \cdot g, f \cdot h\,] \tag{1.39}$$

$+$-absorption :

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[d, "i"'] & A+B \arrow[d, "i+j"'] & B \arrow[l, "i_2"'] \arrow[d, "j"] \\
D \arrow[r, "i_1"] \arrow[dr, "g"'] & D+E \arrow[d, "{[\,g,h\,]}"] & E \arrow[l, "i_2"] \arrow[dl, "h"] \\
& C &
\end{tikzcd}
\qquad [\,g,h\,] \cdot (i+j) = [\,g \cdot i, h \cdot j\,] \tag{1.40}$$

$+$-functor :

$$(g \cdot h) + (i \cdot j) = (g+i) \cdot (h+j) \tag{1.41}$$

$+$-functor-id :

$$id_A + id_B = id_{A+B} \tag{1.42}$$

In summary, the typing-rules of the *either* and *sum* combinators are as follows:

$$\frac{C \xleftarrow{f} A \qquad D \xleftarrow{g} B}{C + D \xleftarrow{f+g} A + B} \qquad \frac{C \xleftarrow{f} A \qquad C \xleftarrow{g} B}{C \xleftarrow{[\,f,g\,]} A + B} \tag{1.43}$$

Exercise 1.5 By analogy (duality) with $swap$, show that $[\,i_2, i_1\,]$ is its own inverse and so that fact

$$A + B \quad \cong \quad B + A \tag{1.44}$$

holds.
□

Exercise 1.6 Dualize (1.32), that is, write the iso which witnesses fact

$$A + (B + C) \quad \cong \quad (A + B) + C \tag{1.45}$$

from right to left. Use the `either` syntax available from the HUGS *Standard Prelude* to encode this iso in HASKELL.
□

1.10 Mixing products and coproducts

Datatype constructions $A \times B$ and $A + B$ have been introduced above as devices required for expressing the codomain of *splits* ($A \times B$) or the domain of *eithers* ($A + B$). Therefore, a function mapping values of a coproduct (say $A + B$) to values of a product (say $A' \times B'$) can be expressed alternatively as an *either* or as a *split*. In the first case, both components of the *either* combinator are *splits*. In the latter, both components of the *split* combinator are *eithers*.

This exchange of format in defining such functions is known as the *exchange law*. It states the functional equality which follows:

$$[\,\langle f, g\rangle, \langle h, k\rangle\,] \quad = \quad \langle [\,f, h\,], [\,g, k\,]\rangle \tag{1.46}$$

It can be checked by type-inference that both the left-hand side and the right-hand side expressions of this equality have type $B \times D \longleftarrow A + C$, for $B \xleftarrow{f} A$, $D \xleftarrow{g} A$, $B \xleftarrow{h} C$ and $D \xleftarrow{k} C$.

An example of a function which is in the exchange-law format is isomorphism

$$A \times (B + C) \xleftarrow{undistr} (A \times B) + (A \times C) \tag{1.47}$$

(pronounce $undistr$ as "un-distribute-right") which is defined by

$$undistr \quad \stackrel{\rm def}{=} \quad [\,id \times i_1, id \times i_2\,] \tag{1.48}$$

and witnesses the fact that product distributes through coproduct:

$$A \times (B + C) \quad \cong \quad (A \times B) + (A \times C) \tag{1.49}$$

In this context, suppose that we know of three functions $D \xleftarrow{f} A$, $E \xleftarrow{g} B$ and $F \xleftarrow{h} C$. By (1.43) we infer $E + F \xleftarrow{g+h} B + C$. Then, by (1.22) we infer

$$D \times (E + F) \xleftarrow{f \times (g+h)} A \times (B + C) \tag{1.50}$$

So, it makes sense to combine products and sums of functions and the expressions which denote such combinations have the same "shape" (or symbolic pattern) as the expressions which denote their domain and range — the $\ldots \times (\cdots + \cdots)$ "shape" in this example. In fact, if we *abstract* such a pattern via some symbol, say F — that is, if we define

$$\mathsf{F}(\alpha, \beta, \gamma) \quad \stackrel{\mathrm{def}}{=} \quad \alpha \times (\beta + \gamma)$$

— then we can write $\mathsf{F}(D, E, F) \xleftarrow{\mathsf{F}(f,g,h)} \mathsf{F}(A, B, C)$ for (1.50).

This kind of abstraction works for every combination of products and coproducts. For instance, if we now abstract the right-hand side of (1.47) via pattern

$$\mathsf{G}(\alpha, \beta, \gamma) \quad \stackrel{\mathrm{def}}{=} \quad (\alpha \times \beta) + (\alpha \times \gamma)$$

we have $\mathsf{G}(f, g, h) = (f \times g) + (f \times h)$, a function which maps $\mathsf{G}(A, B, C) = (A \times B) + (A \times C)$ onto $\mathsf{G}(D, E, F) = (D \times E) + (D \times F)$. All this can be put in a diagram

$$\begin{tikzcd}
\mathsf{F}(A,B,C) \arrow[d, "\mathsf{F}(f,g,h)"'] & \mathsf{G}(A,B,C) \arrow[l, "undistr"'] \arrow[d, "\mathsf{G}(f,g,h)"] \\
\mathsf{F}(D,E,F) & \mathsf{G}(D,E,F)
\end{tikzcd}$$

which unfolds to

$$\begin{tikzcd}
A \times (B + C) \arrow[d, "f \times (g+h)"'] & (A \times B) + (A \times C) \arrow[l, "undistr"'] \arrow[d, "(f \times g)+(f \times h)"] \\
D \times (E + F) & (D \times E) + (D \times F)
\end{tikzcd} \tag{1.51}$$

once the F and G patterns are instantiated. Some interesting issues which stem from (completing) this diagram will be discussed in the next section.

Exercise 1.7 Apply the *exchange law* to $undistr$.
□

Exercise 1.8 Complete the "?"s in diagram

$$\begin{tikzcd} & ? \arrow[dl, "{[x,y]}"'] \arrow[d, "id+id\times f"] \\ ? & ? \arrow[l, "{[k,g]}"] \end{tikzcd}$$

and then solve the implicit equation for x and y.

□

Exercise 1.9 Repeat exercise 1.8 with respect to diagram

$$\begin{tikzcd} ? \arrow[r, "h+\langle i,j\rangle"] \arrow[dr, "x+y"'] & ? \arrow[d, "id+id\times f"] \\ & ? \end{tikzcd}$$

□

1.11 Natural properties

Let us resume discussion about $undistr$ and the two other functions in diagram (1.51). What about using $undistr$ itself to close this diagram, at the bottom? Note that definition (1.48) works for D, E and F in the same way it does for A, B and C. (Indeed, the choice of the A, B and C was in (1.47) rather arbitrary.) Therefore, we get:

$$\begin{tikzcd} A \times (B + C) \arrow[d, "f\times(g+h)"'] & (A \times B) + (A \times C) \arrow[l, "undistr"'] \arrow[d, "(f\times g)+(f\times h)"] \\ D \times (E + F) & (D \times E) + (D \times F) \arrow[l, "undistr"] \end{tikzcd}$$

which expresses a very important property of $undistr$:

$$(f \times (g + h)) \cdot undistr \quad = \quad undistr \cdot ((f \times g) + (f \times h)) \tag{1.52}$$

This is called the *natural* property of $undistr$. This kind of property (often called *free* instead of *natural*) is not a privilege of $undistr$. As a matter of fact, every function interfacing patterns such as F or G above will exhibit its own *natural* property. Furthermore, we have already quoted *natural* properties without mentioning it. Recall (1.10), for instance. This property (establishing id as the *unit* of composition) is, after all, the *natural* property of id. In this case we have $\mathsf{F}\,\alpha = \mathsf{G}\,\alpha = \alpha$, as can be easily observed in diagram (1.11).

In general, *natural* properties are described by diagrams in which two "copies" of the operator of interest are drawn as horizontal arrows:

$$
\begin{tikzcd}
A \arrow[d, "f"'] \\
B
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{F}\,A \arrow[d, "\mathsf{F}\,f"'] & \mathsf{G}\,A \arrow[l, "\phi"'] \arrow[d, "\mathsf{G}\,f"] \\
\mathsf{F}\,B & \mathsf{G}\,B \arrow[l, "\phi"]
\end{tikzcd}
\qquad
(\mathsf{F}\,f) \cdot \phi = \phi \cdot (\mathsf{G}\,f) \tag{1.53}
$$

Note that f is universally quantified, that is to say, the *natural* property holds for every $f : B \longleftarrow A$.

Diagram (1.53) corresponds to unary patterns F and G. As we have seen with $undistr$, other functions (g,h *etc.*) come into play for multiary patterns. A very important rôle will be assigned throughout this book to these F,G, *etc.* "shapes" or patterns which are shared by pointfree functional expressions and by their domain and codomain expressions. From chapter 2 onwards we will refer to them by their proper name — "functor" — which is standard in mathematics and computer science. Then we will also explain the names assigned to properties such as, for instance, (1.27) or (1.41).

Exercise 1.10 Show that (1.25) and (1.26) are *natural* properties. Dualize these properties. **Hint**: recall diagram (1.40).
□

Exercise 1.11 Establish the *natural* properties of the $swap$ (1.30) and $assocr$ (1.32) isomorphisms.
□

1.12 Guards and McCarthy's conditional

Most functional programming languages and notations cater for pointwise conditional expressions of the form

$$if \ \ (p\,x) \ \ then \ \ (g\,x) \ \ else \ \ (h\,x)$$

meaning

$$\left\{ \begin{array}{rcl} p\,x & \Rightarrow & g\,x \\ \neg(p\,x) & \Rightarrow & h\,x \end{array} \right.$$

for some given predicate $\mathsf{Bool} \xleftarrow{p} A$, some "then"-function $B \xleftarrow{g} A$ and some "else"-function $B \xleftarrow{h} A$. Bool is the primitive datatype containing truth values FALSE and TRUE.

Can such expressions be written in pointfree-style? They can, provided we introduce the so-called "McCarthy conditional" functional form

$$p \rightarrow g, h$$

which is defined by

$$p \rightarrow g, h \quad \stackrel{\text{def}}{=} \quad [\, g, h\,] \cdot p? \tag{1.54}$$

a definition we can understand provided we know the meaning of the "$p?$" construct. We call $A + A \xleftarrow{p?} A$ a *guard*, or better, the guard associated to a given predicate $\mathsf{Bool} \xleftarrow{p} A$. Every predicate p gives birth to its own guard $p?$ which, at pointwise-level, is defined as follows:

$$(p?)a \quad = \quad \begin{cases} \quad p\,a & \Rightarrow \quad i_1\,a \\ \neg(p\,a) & \Rightarrow \quad i_2\,a \end{cases} \tag{1.55}$$

In a sense, guard $p?$ is more "informative" than p alone: it provides information about the outcome of testing p on some input a, encoded in terms of the coproduct injections (i_1 for a true outcome and i_2 for a false outcome, respectively) without losing the input a itself.

The following fact, which we will refer to as *McCarthy's conditional fusion law*, is a consequence of $+$-fusion (1.39):

$$f \cdot (p \rightarrow g, h) \quad = \quad p \rightarrow f \cdot g, f \cdot h \tag{1.56}$$

We shall introduce and define instances of predicate p as long as they are needed. A particularly important assumption of our notation should, however, be mentioned at this point: we assume that, for every datatype A, the equality predicate $\mathsf{Bool} \xleftarrow{=_A} A \times A$ is defined in a way which guarantees three basic properties: reflexivity ($a =_A a$ for every a), transitivity ($a =_A b$ and $b =_A c$ implies $a =_A c$) and symmetry ($a =_A b$ iff $b =_A a$). Subscript A in $=_A$ will be dropped wherever implicit in the context.

In HASKELL programming the equality predicate for a type becomes available by declaring the type as an instance of class `Eq`, which exports equality predicate `(==)`. This does not, however, guarantee the reflexive, transitive and symmetry properties, which need to be proved by dedicated mathematical arguments.

Exercise 1.12 Prove that the following equality between two conditional expressions

$$\begin{array}{rl} & k(if\ \ p\,x\ \ then\ \ f\,x\ \ else\ \ h\,x, if\ \ p\,x\ \ then\ \ g\,x\ \ else\ \ i\,x) \\ = & if\ \ p\,x\ \ then\ \ k(f\,x, g\,x)\ \ else\ \ k(h\,x, i\,x) \end{array}$$

holds by rewriting it in the pointfree style (using the McCarthy's conditional combinator) and applying the *exchange law* (1.46), among others.
□

Exercise 1.13 Prove law (1.56).
□

1.13 Gluing functions which do not compose — exponentials

Now that we have made the distinction between the pointfree and pointwise functional notations reasonably clear, it is instructive to revisit section 1.2 and identify *functional application* as the "bridge" between the pointfree and pointwise worlds. However, we should say "a bridge" rather than "the bridge", for in this section we enrich such an interface with another "bridge" which is very relevant to programming.

Suppose we are given the task to combine two functions $B \xleftarrow{f} C \times A$ and $D \xleftarrow{g} A$. It is clear that none of the combinations $f \cdot g$, $\langle f, g \rangle$ or $[\, f, g\,]$ is well-typed. So, f and g cannot be put together directly — they require some extra interfacing.

Note that $\langle f, g \rangle$ would be well-defined in case the C component of f's domain could be somehow "ignored". Suppose, in fact, that in some particular context the first argument of f happens to be "irrelevant", or to be frozen to some $c \in C$. It is easy to derive a new function

$$
\begin{array}{rcl}
f_c & : & A \longrightarrow B \\
f_c\, a & \stackrel{\mathrm{def}}{=} & f(c, a)
\end{array}
$$

from f which combines nicely with g via the *split* combinator: $\langle f_c, g \rangle$ is well-defined and bears type $B \times D \longleftarrow A$. For instance, suppose that $C = A$ and f is the equality predicate $=$ on A. Then $\mathsf{Bool} \xleftarrow{=_c} A$ is the "equal to c" predicate on A values:

$$
=_c\; a \quad \stackrel{\mathrm{def}}{=} \quad a = c \tag{1.57}
$$

As another example, recall function $twice$ (1.3) which could be defined as $\times_2$ using the new notation.

However, we need to be more careful about what is meant by f_c. Such as functional application, expression f_c interfaces the pointfree and the pointwise levels — a function (f) and a value (c). But there is a major distinction between $f\, c$ and f_c — while the former denotes a value of type B, *i.e.* $f\, c \in B$, f_c denotes a function of type $B \longleftarrow A$. We will say that $f_c \in B^A$ by introducing a new datatype construct which we will call the *exponential*:

$$
B^A \quad \stackrel{\mathrm{def}}{=} \quad \{ f \mid f : B \longleftarrow A \} \tag{1.58}
$$

There are strong reasons to adopt the B^A notation in detriment of the more obvious $B \leftarrow A$ or $A \rightarrow B$ alternatives, as we shall see shortly.

The B^A exponential datatype is therefore inhabited by functions from A to B (that is to say, functional declaration $f : B \longleftarrow A$ means the same as $f \in B^A$). And what do we want functions for? We want to apply them. So it is natural to introduce the *apply* operator

$$
\begin{aligned}
ap :\ & B \xleftarrow{ap} B^A \times A \\
ap(f, a) & \stackrel{\mathrm{def}}{=} f\, a
\end{aligned}
$$

which applies a function f to an argument a.

Back to generic binary function $B \xleftarrow{f} C \times A$, let us now think of the operation which, for every $c \in C$, produces $f_c \in B^A$. This can be regarded as a function of signature $B^A \longleftarrow C$ which expresses f as a kind of C-indexed family of functions of signature $A \longleftarrow B$. We will denote such a function by $\overline{f}$ (read $\overline{f}$ as "f transposed"). Intuitively, we want f and $\overline{f}$ to be related to each other by the following property:

$$
f(c, a) \quad = \quad (\overline{f}\, c)a \tag{1.59}
$$

Both functions compute the same value. But, in a sense, $\overline{f}$ is more tolerant than f: while the latter is binary and requires *both* arguments (c, a) to become available before application, the former is happy to be provided with c first and with a later on, if actually required by the evaluation process.

Similarly to $A \times B$ and $A + B$, exponential B^A exhibits it own laws for cancellation, reflexion and fusion:

Exponentials cancellation :

$$
\begin{tikzcd}
B^A & B^A \times A \arrow[r, "ap"] & B \\
C \arrow[u, "\overline{f}"] & C \times A \arrow[u, "\overline{f} \times id"] \arrow[ur, "f"'] &
\end{tikzcd}
\qquad f = ap \cdot (\overline{f} \times id) \tag{1.60}
$$

Exponentials reflexion :

$$
\begin{tikzcd}
B^A & B^A \times A \arrow[r, "ap"] & B \\
B^A \arrow[u, "id_{B^A}"] & B^A \times A \arrow[u, "id_{B^A} \times id_A"] \arrow[ur, "ap"'] &
\end{tikzcd}
\qquad \overline{ap} = id_{B^A} \tag{1.61}
$$

Exponentials fusion :

$$
\begin{tikzcd}
B^A & B^A \times A \arrow[r, "ap"] & B \\
C \arrow[u, "\overline{g}"] & C \times A \arrow[u, "\overline{g} \times id"] \arrow[ur, "g"] & \\
D \arrow[u, "f"] & D \times A \arrow[u, "f \times id"] \arrow[uur, "g \cdot (f \times id)"'] &
\end{tikzcd}
\qquad \overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{1.62}
$$

Note that the cancellation law is nothing but fact (1.59) written in the pointfree style.

In order to present the absorption law for exponentials we need to introduce a new functional combinator which we will write as f^A. Its type-rule is as follows:

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

Given A and $C \xleftarrow{f} B$, f^A is the function which accepts some input function $B \xleftarrow{g} A$ as argument and produces function $f \cdot g$ as result. So f^A is the "compose with f" functional combinator:

$$(f^A)g \stackrel{\mathrm{def}}{=} f \cdot g \tag{1.63}$$

Now we are ready to understand the laws which follow:

Exponentials absorption :

$$\begin{tikzcd}
D^A & & D^A \times A \arrow[r, "ap"] & D \\
B^A \arrow[u, "f^A"] & & B^A \times A \arrow[u, "f^A \times id"] \arrow[r, "ap"] & B \arrow[u, "f"'] \\
C \arrow[u, "\overline{g}"] & & C \times A \arrow[u, "\overline{g} \times id"] \arrow[ur, "g"'] &
\end{tikzcd}
\qquad \overline{f \cdot g} = f^A \cdot \overline{g} \tag{1.64}$$

Exponentials-functor :

$$(g \cdot h)^A = g^A \cdot h^A \tag{1.65}$$

Exponentials-functor-id :

$$id^A = id \tag{1.66}$$

To conclude this section we need to explain why we have adopted the apparently esoteric B^A notation for the "function from A to B" data type. Let us introduce the following operator

$$\mathit{curry}\ f \stackrel{\mathrm{def}}{=} \overline{f} \tag{1.67}$$

which maps a function f to its transpose $\overline{f}$. This operator, which is very familiar to functional programmers, maps functions in some function space $B^{C \times A}$ to functions in $(B^A)^C$. Its inverse (known as the *uncurry* function) also exists. In the HUGS *Standard Prelude* we find them declared as follows:

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

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

From (1.67) it is obvious see that writing $\overline{f}$ or *curry* f is a matter of taste, the latter being more in the tradition of functional programming. For instance, the fusion law (1.62) can be re-written into

$$\mathit{curry}\,(g \cdot (f \times id)) = \mathit{curry}\,g \cdot f$$

etc.

It is known from mathematics that *curry* and *uncurry* are isos witnessing the following isomorphism which is at the core of the theory of functional programming:

$$B^{C \times A} \quad \cong \quad \left(B^A\right)^C \tag{1.68}$$

Fact (1.68) clearly resembles a well known equality concerning numeric exponentials, $b^{c \times a} = \left(b^a\right)^c$. But other known facts about numeric exponentials, *e.g.* $a^{b+c} = a^b \times a^c$ or $(b \times c)^a = b^a \times c^a$ find their counterpart in functional exponentials. The counterpart of the former,

$$A^{B+C} \quad \cong \quad A^B \times A^C \tag{1.69}$$

arises from the uniqueness of the *either* combination: every pair of functions $(f, g) \in A^B \times A^C$ leads to a unique function $[\,f, g\,] \in A^{B+C}$ and vice-versa, every function in A^{B+C} is the *either* of some function in A^B and of another in A^C.

The function exponentials counterpart of the second fact about numeric exponentials above is

$$(B \times C)^A \cong B^A \times C^A \tag{1.70}$$

This can be justified by a similar argument concerning the uniqueness of the *split* combinator $\langle f, g \rangle$.

What about other facts valid for numeric exponentials such as $a^0 = 1$ and $1^a = 1$? We need to know what 0 and 1 mean as datatypes. Such elementary datatypes are presented in the section which follows.

Exercise 1.14 Load module `Set.hs` (*cf.* section A.1) into the HUGS interpreter and check the types assigned to the following functional expressions:

```
curry ap
\f -> ap . ( f >< id)
uncurry . curry
```

Which of these is functionally equivalent to the `uncurry` function and why? Which of these are functionally equivalent to identity functions? Justify.

□

1.14 Elementary datatypes

So far we have talked mostly about arbitrary datatypes represented by capital letters A, B, *etc.*(lowercase `a`, `b`, *etc.* in the HASKELL illustrations). We also mentioned $\mathbb{R}$, Bool and $\mathbb{N}$ and, in particular, the fact that we can associate to each natural number n its *initial segment* $\mathsf{n} = \{1, 2, \ldots, n\}$. We extend this to $\mathbb{N}_0$ by stating $\mathsf{0} = \{\}$ and, for $n > 0$, $\mathsf{n+1} = \{n + 1\} \cup \mathsf{n}$.

Initial segments can be identified with enumerated types and are regarded as primitive datatypes in our notation. We adopt the convention that primitive datatypes are written in the *sans serif* font and so, strictly speaking, n is distinct from n: the latter denotes a natural number while the former denotes a datatype.

Datatype 0

Among such enumerated types, $\mathsf{0}$ is the smallest because it is empty. This is the `Void` datatype in HASKELL, which has no constructor at all. Datatype $\mathsf{0}$ (which we tend to write simply as 0) may not seem very "useful" in practice but it is of theoretical interest. For instance, it is easy to check that the following "obvious" properties hold:

$$A + 0 \cong A \tag{1.71}$$

$$A \times 0 \cong 0 \tag{1.72}$$

Datatype 1

Next in the sequence of initial segments we find 1, which is singleton set $\{1\}$. How useful is this datatype? Note that every datatype A containing exactly one element is isomorphic to $\{1\}$, *e.g.* $A = \{\text{NIL}\}$, $A = \{0\}$, $A = \{1\}$, $A = \{\text{FALSE}\}$, *etc.*. We represent this class of singleton types by 1.

Recall that isomorphic datatypes have the same expressive power and so are "abstractly identical". So, the actual choice of inhabitant for datatype 1 is irrelevant, and we can replace any particular singleton set by another without losing information. This is evident from the following relevant facts involving 1:

$$A \times 1 \cong A \tag{1.73}$$

$$A^0 \cong 1 \tag{1.74}$$

We can read (1.73) informally as follows: if the second component of a "struct" cannot change then it is useless and can be ignored. Selector π_1 is, in this context, an iso mapping the left-hand side of (1.73) to its right-hand side. Its inverse is $\langle id, \underline{c} \rangle$ where c is a particular choice of inhabitant for datatype 1. Concerning (1.74), A^0 denotes the set of all functions from the empty set to some A. What does (1.74) mean? It simply tells us that there is only one function in such a set — the empty function mapping "no" value at all. This fact confirms our choice of notation once again (compare with $a^0 = 1$ in a numeric context).

Next, we may wonder about facts

$$1^A \cong 1 \tag{1.75}$$

$$A^1 \cong A \tag{1.76}$$

which are the functional exponentiation counterparts of $1^a = 1$ and $a^1 = a$. Fact (1.75) is valid: it means that there is only one function mapping A into some singleton set $\{c\}$ — the constant function $\underline{c}$. There is no room for another function in 1^A because only c is available as output value. Fact (1.76) is also valid: all functions in A^1 are constant functions and there are as many constant functions in such a set as there are elements in A.

In summary, when referring to datatype 1 we will mean an arbitrary singleton type, and there is a unique iso (and its inverse) between two such singleton types. The HASKELL representative of 1 is datatype `()`, called the *unit type*, which contains exactly constructor `()`. It may seem confusing to denote the type and its unique inhabitant by the same symbol but it is not, since HASKELL keeps track of types and constructors in separate symbol sets.

Finally, what can we say about $1 + A$? Every function $B \xleftarrow{f} 1 + A$ observing this type is bound to be an *either* $[\,\underline{b_0}, g\,]$ for $b_0 \in B$ and $B \xleftarrow{g} A$. This is very similar to the handling of a pointer in C or PASCAL: we "pull a rope" and either we get nothing (1) or we get something useful of type A. In programming terms "nothing" above means a predefined value NIL. This analogy supports our preference in the sequel for NIL as canonical inhabitant of datatype 1. In fact, we will refer to $1 + A$ (or $A + 1$) as the "pointer to A" datatype. This corresponds to the `Maybe` type constructor of the HUGS *Standard Prelude*.

Datatype 2

Let us inspect the $1+1$ instance of the "pointer" construction mentioned above. Any observation $B \xleftarrow{f} 1 + 1$ can be decomposed in two constant functions: $f = [\,\underline{b_1}, \underline{b_2}\,]$. Now suppose that $B = \{b_1, b_2\}$ (for $b_1 \neq b_2$). Then $1 + 1 \cong B$ will hold, for whatever choice of inhabitants b_1 and b_2. So we are in a situation similar to 1: we will use symbol 2 to represent the abstract class of all such Bs containing exactly two elements. Therefore, we can write:

$$1 + 1 \cong 2$$

Of course, $\mathsf{Bool} = \{\text{TRUE}, \text{FALSE}\}$ and initial segment $2 = \{1, 2\}$ are in this abstract class. In the sequel we will show some preference for the particular choice of inhabitants $b_1 = \text{TRUE}$ and $b_2 = \text{FALSE}$, which enables us to use symbol 2 in places where Bool is expected.

Exercise 1.15 Relate HASKELL expressions

```
either (split (const True) id) (split (const False) id)
```

and

```
\f->(f True, f False)
```

to the following isomorphisms involving generic elementary type 2:

$$2 \times A \quad \cong \quad A + A \tag{1.77}$$
$$A \times A \quad \cong \quad A^2 \tag{1.78}$$

Apply the *exchange law* (1.46) to the first expression above.
□

1.15 Finitary products and coproducts

In section 1.8 it was suggested that product could be regarded as the abstraction behind data-structuring primitives such as `struct` in C or `record` in PASCAL. Similarly, coproducts were suggested in section 1.9 as abstract counterparts of C unions or PASCAL variant records. For a finite A, exponential B^A could be realized as an *array* in any of these languages. These analogies are captured in table 1.1.

In the same way C `structs` and `unions` may contain finitely many entries, as may PASCAL (variant) records, product $A \times B$ extends to finitary product $A_1 \times \ldots \times A_n$, for $n \in \mathbb{N}$, also denoted by $\Pi_{i=1}^n A_i$, to which as many projections π_i are associated as the number n of involved factors. Of course, *splits* become n-ary as well

$$\langle f_1, \ldots, f_n \rangle : \; A_1 \times \ldots \times A_n \longleftarrow B$$

for $f_i : \; A_i \longleftarrow B$, $i = 1, n$.

Dually, coproduct $A + B$ is extensible to the finitary sum $A_1 + \cdots + A_n$, for $n \in \mathbb{N}$, also denoted by $\sum_{j=1}^n A_j$, to which as many injections i_j are assigned as the number n of involved terms. Similarly, *eithers* become n-ary

$$[\, f_1, \ldots, f_n \,] : \; A_1 + \ldots + A_n \longrightarrow B$$

for $f_i : \; B \longleftarrow A_i$, $i = 1, n$.

Datatype n

Next after 2, we may think of 3 as representing the abstract class of all datatypes containing exactly three elements. Generalizing, we may think of n as representing the abstract class of all datatypes containing exactly n elements. Of course, initial segment n will be in this abstract class. (Recall (1.16), for instance: both $\mathsf{Weekday}$ and $\mathsf{7}$ are abstractly represented by 7.) Therefore,

$$n \quad \cong \quad \underbrace{1 + \cdots + 1}_{n}$$

and

$$\underbrace{A \times \ldots \times A}_{n} \quad \cong \quad A^n \tag{1.79}$$
$$\underbrace{A + \ldots + A}_{n} \quad \cong \quad n \times A \tag{1.80}$$

Abstract notation	PASCAL	C/C++	**Description**

$A \times B$	`record`
	`P: A;`
	`S: B`
	`end;`
		`struct {`
		`A first;`
		`B second;`
		`};`
			Records

$A + B$	`record`
	`case`
	`tag: integer`
	`of x =`
	`1: (P:A);`
	`2: (S:B)`
	`end;`
		`struct {`
		`int tag; /* 1,2 */`
		`union {`
		`A ifA;`
		`B ifB;`
		`} data;`
		`};`
			Variant records

B^A	`array[A] of B`	`B ...[A]`	Arrays
$1 + A$	`^A`	`A *...`	Pointers

Table 1.1: Abstract notation versus programming language data-structures.

hold.

Exercise 1.16 On the basis of table 1.1, encode $undistr$ (1.48) in C or PASCAL. Compare your code with the HASKELL pointfree and pointwise equivalents.
□

1.16 Initial and terminal datatypes

All properties studied for binary *splits* and binary *eithers* extend to the finitary case. For the particular situation $n = 1$, we will have $\langle f \rangle = [\, f \,] = f$ and $\pi_1 = i_1 = id$, of course. For the particular situation $n = 0$, finitary products "degenerate" to 1 and finitary coproducts "degenerate" to 0. So diagrams (1.20) and (1.35) are reduced to

$$\begin{tikzcd} 1 & & 0 \arrow[d, "{[\ ]}"] \\ C \arrow[u, "\langle\rangle"] & & C \end{tikzcd}$$

The standard notation for the empty *split* $\langle\rangle$ is $!_C$, where subscript C can be omitted if implicit in the context. By the way, this precisely the only function in 1^C, recall (1.75). Dually, the standard notation for the empty *either* $[\ ]$ is $?_C$, where subscript C can also be omitted. By the way, this precisely the only function in C^0, recall (1.74).

In summary, we may think of 0 and 1 as, in a sense, the "extremes" of the whole datatype spectrum. For this reason they are called *initial* and *terminal*, respectively. We conclude this subject with the presentation of their main properties which, as we have said, are instances of properties we have stated for products and coproducts.

Initial datatype reflexion :

$$\begin{tikzcd} 0 \arrow[loop above, "?_0 = id_0"] \end{tikzcd} \qquad\qquad ?_0 = id_0 \tag{1.81}$$

Initial datatype fusion :

$$\begin{tikzcd} 0 \arrow[d, "?_A"'] \arrow[dr, "?_B"] & \\ A \arrow[r, "f"'] & B \end{tikzcd} \qquad f \cdot ?_A = ?_B \tag{1.82}$$

Terminal datatype reflexion :

$$\begin{tikzcd} 1 \arrow[loop above, "!_1 = id_1"] \end{tikzcd} \qquad !_1 = id_1 \tag{1.83}$$

Terminal datatype fusion :

$$\begin{tikzcd} 1 & \\ A \arrow[u, "!_A"] & B \arrow[l, "f"] \arrow[ul, "!_B"'] \end{tikzcd} \qquad !_A \cdot f = !_B \tag{1.84}$$

Exercise 1.17 Particularize the *exchange law* (1.46) to empty products and empty coproducts, *i.e.* 1 and 0.
□

1.17 Sums and products in HASKELL

We conclude this chapter with an analysis of the main primitive available in HASKELL for creating datatypes: the `data` declaration. Suppose we declare

```
data CostumerId = P Int | CC Int
```

meaning to say that, for some company, a client is identified either by its passport number or by its credit card number, if any. What does this piece of syntax precisely mean?

If we enquire the HUGS *interpreter* about what he knows about `CostumerId`, the reply will be contain the following information:

```
Main> :i CostumerId
-- type constructor
data CostumerId

-- constructors:
P :: Int -> CostumerId
CC :: Int -> CostumerId
```

In general, let A and B be two known datatypes. Via declaration

```
data C = C1 A | C2 B
```

one obtains from HUGS a new datatype C equipped with constructors $C \xleftarrow{C1} A$ and $C \xleftarrow{C1} B$, in fact the only ones available for constructing values of C:

$$\begin{tikzcd}
A \arrow[dr, "C1"'] & & B \arrow[dl, "C2"] \\
& C &
\end{tikzcd}$$

This diagram leads to an obvious instance of coproduct diagram (1.35),

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "C1"'] & A + B \arrow[d, "{[\, C1, C2 \,]}"] & B \arrow[l, "i_2"'] \arrow[dl, "C2"] \\
& C &
\end{tikzcd}$$

describing that a `data` declaration in HASKELL means the *either* of its constructors.

Because there are no other means to build C data, it follows that C is isomorphic to $A + B$. So $[\, C1, C2 \,]$ has an inverse, say inv, which is such that $inv \cdot [\, C1, C2 \,] = id$. How do we calculate inv? Let us first think of the generic situation of a function $D \xleftarrow{f} C$ which observes datatype C:

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "C1"'] & A + B \arrow[d, "{[\, C1, C2 \,]}"] & B \arrow[l, "i_2"'] \arrow[dl, "C2"] \\
& C \arrow[d, "f"] & \\
& D &
\end{tikzcd}$$

This is an opportunity for $+$-*fusion* (1.39), whereby we obtain

$$f \cdot [\, C1, C2 \,] \quad = \quad [\, f \cdot C1, f \cdot C2 \,]$$

Therefore, the observation will be fully described provided we explain how f behaves with respect to $C1$ — *cf.* $f \cdot C1$ — and with respect to $C2$ — *cf.* $f \cdot C2$. This is what is behind the typical *inductive* structure of pointwise f, which will be made of two and only two clauses:

$$\begin{array}{l}
f : \; C \longrightarrow D \\
f(C1\, a) = \ldots \\
f(C2\, b) = \ldots
\end{array}$$

Let us use this in calculating the inverse inv of $[\,C1, C2\,]$:

$$\begin{array}{cl} & inv \cdot [\,C1, C2\,] = id \\ \leftrightarrow & \qquad \{ \text{ by } +\text{-}\textit{fusion} \text{ (1.39)} \} \\ & [\,inv \cdot C1, inv \cdot C2\,] = id \\ \leftrightarrow & \qquad \{ \text{ by } +\text{-}\textit{reflexion} \text{ (1.38)} \} \\ & [\,inv \cdot C1, inv \cdot C2\,] = [\,i_1, i_2\,] \\ \leftrightarrow & \qquad \{ \textit{ either} \text{ uniqueness} \} \\ & inv \cdot C1 = i_1 \wedge inv \cdot C2 = i_2 \end{array}$$

Therefore:

$$\begin{array}{l} inv : \ C \longrightarrow A + B \\ inv(C1\,a) = i_1\,a \\ inv(C2\,a) = i_2\,b \end{array}$$

In summary, $C1$ is a "renaming" of injection i_1 and $C2$ is a "renaming" of injection i_2.

What is the pointer-equivalent in HASKELL? This corresponds to $A = 1$ above and to the following HASKELL declaration:

```
data C = C1 () | C2 B
```

Note that HASKELL allows for a more programming-oriented alternative in this case, in which the unit type `()` is eliminated:

```
data C = C1 | C2 B
```

The difference is that here `C1` denotes an inhabitant of C (and so a clause $f(C1\,a) = \ldots$ is rewritten to $f\,C1 = \ldots$) while above `C1` denotes a (constant) function $C \xleftarrow{C1} 1$. Isomorphism (1.76) helps in comparing these two alternative situations.

Exercise 1.18 Establish the difference between the following two declarations in HASKELL,

```
data D = D1 A | D2 B C
```

and

```
data E = E1 A | E2 (B,C)
```

for `A`, `B` and `C` any three predefined types. Are D and E isomorphic? If so, can you specify and encode the corresponding isomorphism?

□

1.18 Bibliography notes

Almost two decades ago John Backus read, in his Turing Award Lecture, a revolutionary paper [Bac78]. This paper proclaimed conventional command-oriented programming languages obsolete because of their inefficiency arising from retaining, at a high-level, the so-called "memory access bottleneck" of the underlying computation model — the well-known *von Neumann* architecture. Alternatively, the (at the time already existing) *functional programming* style was put forward for two main reasons. Firstly, because of its potential for concurrent and parallel computation. Secondly — and Backus emphasis was really put on this —, because of its strong algebraic basis.

Backus *algebra of (functional) programs* was providential in alerting computer programmers that *computer languages* alone are insufficient, and that only languages which exhibit an *algebra* for reasoning about the objects they purport to describe will be useful in the long run.

The impact of Backus first argument in the computing science and computer architecture communities was considerable, in particular if assessed in quality rather than quantity and in addition to the almost contemporary *structured programming* trend [2]. By contrast, his second argument for changing computer programming was by and large ignored. In fact, only the so-called *algebra of programming* research minorities pursued in this direction. However, the advances in this area throughout the last two decades are impressive and can be fully appreciated by reading a textbook written relatively recently by Bird and de Moor [BdM97]. A comprehensive review of the voluminous literature available in this area can also be found in this book.

Although the need for a pointfree algebra of programming was first identified by Backus, perhaps influenced by Iverson's APL growing popularity in the USA at that time, the idea of reasoning and using mathematics to transform programs is much older and can be traced to the times of McCarthy's work on the foundations of computer programming [McC63], of Floyd's work on program meaning [Flo67] and of Paterson and Hewitt's *comparative schematology* [PH70]. Work of the so-called *program transformation* school was already very expressive in the mid 1970s, see for instance references [BD77].

The mathematics adequate for the effective integration of these related but independent lines of thought was provided by the categorial approach of Manes and Arbib compiled in a textbook [MA86] which has very strongly influenced the last decade of 20th century theoretical computer science.

A so-called MPC ("Mathematics of Program Construction") community has been among the most active in producing an integrated body of knowledge on the algebra of programming which has found in functional programming an eloquent an paradigmatic medium. Functional programming has a tradition of absorbing fresh results from theoretical computer science, algebra and category theory. Languages such as HASKELL [Bir98] have been competing to integrate the most recent developments and therefore are excellent *prototyping* vehicles in courses on program calculation, as happens with this book.

---

[2] Even the C programming language and the UNIX operating system, with their subtle functional flavour, may be regarded as eloquent outcomes of the "going functional" trend.