Deriving Programming Laws Categorically

Nico Verwer

RUU-CS-90-6
February 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

ISSN:0924-3275

Deriving Programming Laws Categorically

Nico Verwer
Department of Computer Science, University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
email: verwer@cs.ruu.nl

February 19, 1990

Abstract

In this article, category theory is used to describe functional programming, with the aim of deriving programming laws. Because category theory is a rather abstract model, it allows us to start from very few principles. As a result, we are able to derive laws which capture a number of laws which were previously derived seperately.

We review some category theoretical concepts, using notions taken over from functional programming. Then we study function spaces, and derive the laws that come with the *lifting* functor. In the last part we show that the polymorphic functions that come with *strong* type-constructors are natural.

Contents

1 Introduction

Programming laws are the basis of the formal derivation of programs from specifications. Such laws can be used to transform an inefficient specification step-by-step into an efficient implementation [3]. At present, the number of laws used has become very large, and some unification of similar laws would be advantageous.

In this article we start from as few principles as possible, in order to obtain a few, general laws. Most of the laws in this article were known previously, but they were *postulated*, and not *derived* from other (more fundamental) laws [1]. A derivation of these laws can increase our understanding of them. To mark the distinction, it is shown clearly which formulas are axioms (numbered A1, A2, ...) and which are derived (numbered D1, D2, ...).

Because the laws of functional programming are statements *about* programs, we need to take an abstract approach to programs. We claim that category theory is a useful framework for the development of a theory of functional programming. This framework has already been explored by other researchers, who have studied specific data types, such as lists, with the help of category theory [6, 9]. In this paper we consider the basic building blocks of functional programs, (polymorphic) functions. We use the category of sets as a model for for functional programming languages, although we do not explicitly give a semantics.

Most of the results we derive were known before, either in category theory or in functional programming. The main contribution of this paper is the method of deriving programming laws categorically. In another paper, we have used this method to derive laws which hold for *homomorphisms* on data types [10].

Our treatment is aimed at those who have not studied category theory in great detail, and we shall use 'functional programming parlance' as much as possible. However, we highly recommend reading [8], which is a good introduction to category theory, and [2], which shows how category theory may be applied to functional programming.

2 Categorical background

2.1 Types and functions — Objects and arrows

Since we want to derive laws about programs, which are independent of the data manipulated by these programs, we shall not consider the inner structure of datatypes. If we look at datatypes only from the outside, they are *objects* in a *category*, without any further structure. The particular category in which we model functional programming is the category of sets, which is a useful model of the lambda-calculus [4]. (In fact, for most of the results we could have used any cartesian closed category, except in the proof of D17 and section 4.3.)

We shall denote types by $A, B, \ldots$ These may stand for any available datatype in a particular programming language. Functions are represented by *arrows* between types:

$$f : A \to B \quad \text{or} \quad A \xrightarrow{f} B$$

is a total function named f which maps values in A to values in B.

We assume the existence of the identity function

$$A \xrightarrow{i_A} A$$

for every type A. We use $g \circ f$ for the function composition of g and f. This is an associative operation. Type and functions, together with identities and an associative composition operator, form a category.

In order to have functions with more than one argument, we introduce the cartesian product:

Definition 1 Let A and B be types, then there exists a type $A \times B$, the cartesian product of A and B, and there are projection functions

$$A \times B \xrightarrow{\pi} A \quad \text{and} \quad A \times B \xrightarrow{\pi'} B$$

and for every pair of functions

$$C \xrightarrow{f} A \quad \text{and} \quad C \xrightarrow{g} B$$

a unique function

$$C \xrightarrow{\langle f , g \rangle} A \times B$$

such that:

$$f = \pi \circ \langle f , g \rangle \quad g = \pi' \circ \langle f , g \rangle. \tag{A1}$$

□

From this definition, it is easy to prove that

$$\langle f , g \rangle \circ h = \langle f \circ h , g \circ h \rangle. \tag{D1}$$

Some people prefer *commuting diagrams* like the one in figure 1 instead of algebraic equations to express equalities. The dotted arrow indicates that $\langle f , g \rangle$ is the *unique* function such that the diagram commutes. Such diagrams have the advantage that the types of the functions involved can easily be seen, whereas an algebraic equation with full type information can be hard to read. However, the steps of a derivation are clearer from a list of algebraic equations than a single diagram.

The types $A \times (B \times C)$ and $(A \times B) \times C$ are *isomorphic*, which means that there is a function

$$\langle \langle \pi , \pi \circ \pi' \rangle , \pi' \circ \pi' \rangle : A \times (B \times C) \to (A \times B) \times C$$

which has an inverse. Isomorphism is indicated with $\cong$. This means that that $\times$ is associative *up to isomorphism*, so we can make ternary, quaternary, etc. functions, like

$$\text{if} : (\text{bool} \times A \times A) \to A.$$

A binary operator $\oplus$ which takes its arguments from A and B, and yields a result in C is represented by

$$A \times B \xrightarrow{\oplus} C.$$

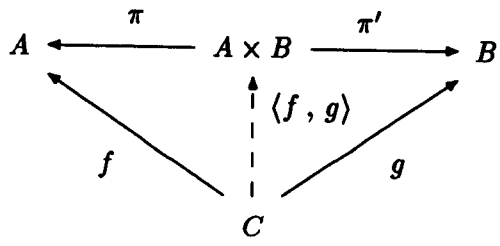

Figure 1: Cartesian product

A constant is a function taking zero parameters. We represent an A-constant c by

$$\mathbf{1} \xrightarrow{c} A$$

where $\mathbf{1}$ is the one-point type, a type with only one value (the *terminal object* in category theory). For every type A there is exactly one arrow from A to $\mathbf{1}$:

$$A \xrightarrow{!_A} \mathbf{1}$$

which maps every value from A to the single value in $\mathbf{1}$. The one-point type is the unit element of the cartesian product:

$$A \times \mathbf{1} \cong \mathbf{1} \times A \cong A. \tag{D2}$$

This is, again, an equality 'up to isomorphism'; one may prove this by asserting that $\pi_{A,\mathbf{1}}$ and $\langle \mathrm{i}_A , !_A \rangle$ are inverses.

2.2 Type constructors — Functors

In category theory, type constructors are modeled by *functors*. Functors extend the notion of type constructors, as they can be applied to both types and functions. For instance, the (bi-)functor $- \times -$ applied to the types A and B gives the cartesian product $A \times B$, and for all functions

$$A \xrightarrow{f} A' \quad \text{and} \quad B \xrightarrow{g} B'$$

we have a function

$$A \times B \xrightarrow{f \times g} A' \times B'$$

satisfying

$$\begin{aligned} \pi \circ (f \times g) &= f \circ \pi \\ \pi' \circ (f \times g) &= g \circ \pi' \end{aligned} \tag{A2}$$

(see figure 2). It is an easy exercise to prove that

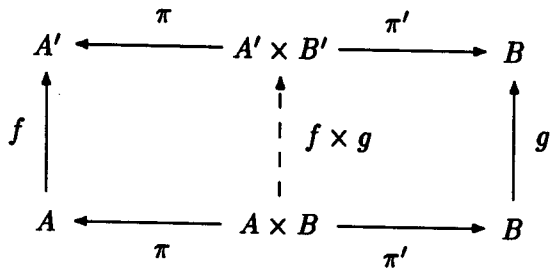

Figure 2: Product of functions

$$f \times g = \langle f \circ \pi \, , \, g \circ \pi' \rangle. \tag{D3}$$

From $1 \times 1 \cong 1$ it follows that for $a : 1 \to A$ and $b : 1 \to B$:

$$a \times b = \langle a \, , \, b \rangle. \tag{D4}$$

Another example of a functor is the list-constructor, named $*$. The type A^* has all (possibly empty) lists with elements of type A. For every $f : A \to B$ there is a function $f^* : A^* \to B^*$, the *map* of f.

Like all functors, the list functor must preserve identity and composition:

$$\begin{aligned} (\mathrm{i}_A)^* &= \mathrm{i}_{A^*} \\ (g \circ f)^* &= g^* \circ f^*. \end{aligned} \tag{A3}$$

For the product functor these properties amount to:

$$\mathrm{i}_A \times \mathrm{i}_B = \mathrm{i}_{A \times B}$$

$$(f \circ f') \times (g \circ g') = (f \times g) \circ (f' \times g').$$

Another important functor is the identity functor I:

$$A^{\mathrm{I}} = A$$

$$f^{\mathrm{I}} = f.$$

Following [2] we shall write functors in postfix-notation where this is not too inconvenient. We do not explicitly indicate functor composition:

$$A^{\dagger\ddagger} = (A^{\dagger})^{\ddagger}.$$

2.3 Polymorphic functions — Natural transformations

The identity $\mathrm{i}_A : A \to A$ is an instance of a function which is polymorphic in A. By polymorphism we mean *parametric* polymorphism, i.e. a polymorphic function operates on structures rather than the types over which these structures are built.

The identity is a polymorphic function from the 'identity structure' to itself:

$$\mathrm{i} : \mathrm{I} \xrightarrow{\cdot} \mathrm{I} \quad \text{or} \quad \mathrm{I} \xrightarrow{\mathrm{i}} \mathrm{I}.$$

In category theory, a polymorphic function is modeled by a *natural transformation*, indicated by the dot under the arrow. The domain and range of polymorphic functions are functors (structures), not types. Only when we instantiate a polymorphic function with a type, we obtain a function between types, like

$$\mathrm{i}_A : A^{\mathrm{I}} \to A^{\mathrm{I}}.$$

We could have written polymorphic functions something like

$$\mathrm{i} : \forall A : A^{\mathrm{I}} \to A^{\mathrm{I}}$$

but the type-variable A is a dummy, so we prefer the shorter notation using dotted arrows.

Usually, we do not indicate the type to which a polymorphic function is instantiated, and we write

$$A \xrightarrow{\mathrm{i}} A \quad \text{instead of} \quad A \xrightarrow{\mathrm{i}_A} A$$

when the type A is clear from the context. It is the task of a *type-inference system* to derive this information, and a particular system may or may not allow the programmer to leave out type-information in certain places.

The behaviour of a truly polymorphic function is not dependent on the element types of its source and target structures. For example, a function on lists must behave similarly on a list of integers and a list of characters. Some programming languages offer a function that converts data of different types to a textual representation (e.g. the write procedure in Pascal), but this is not parametrically polymorphic, as it depends on the inner structure of its argument.

The polymorphism condition is expressed in a law which must hold for all natural transformations:

Definition 2 Let $\dagger$ and $\ddagger$ be functors. A family of functions

$$\tau_A : A^{\dagger} \to A^{\ddagger}$$

is a natural transformation, if for all types A, B and functions $f : A \to B$,

$$\tau_B \circ f^{\dagger} = f^{\ddagger} \circ \tau_A \qquad \text{(A4)}$$

(see figure 3). □

An example is the natural transformation

$$\times \xrightarrow{\pi} \ll$$

where $\ll$ is the bi-functor defined by

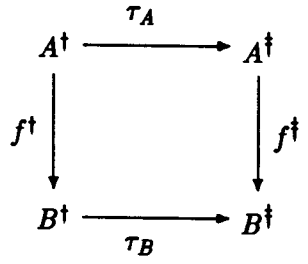

Figure 3: The natural transformation property

$$A \ll B = A$$

$$f \ll g = f.$$

The natural transformation property for this polymorphic function reads

$$\pi_{A',B'} \circ (f \times g) = f \circ \pi_{A,B}.$$

This is exactly the first half of property (A2) of the projection function which we stated before.

Natural transformations can be composed like functions:

$$(\sigma \circ \tau)_A = \sigma_A \circ \tau_A. \tag{A5}$$

Just as we applied functors to functions, we may apply functors to natural transformations:

Definition 3 Let $\tau : \dagger \dot{\longrightarrow} \ddagger$ be a natural transformation, and $\ddagger$ a functor, then

$$\tau^\ddagger : \dagger\ddagger \dot{\longrightarrow} \ddagger\ddagger$$

is a natural transformation, defined by

$$(\tau^\ddagger)_A = (\tau_A)^\ddagger. \tag{A6}$$

□

The fact that polymorphic functions can be modeled by natural transformations was noted before [12]. It was used in [2] to derive some interesting programming laws, and to simplify proofs.

An example of a natural transformation is the function which reverses the order of the elements of a list

$$* \xrightarrow[\cdot]{\text{rev}} *.$$

It is easy to see that this makes the diagram in figure 4 commute. One can apply the list functor to rev, and obtain a natural transformation

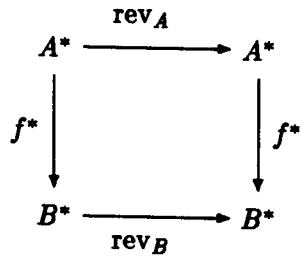

Figure 4: Naturality of rev

$$** \xrightarrow{\mathrm{rev}^*_{\cdot}} **$$

which is the reverse function applied to every element of a list (by A6):

$$(\mathrm{rev}^*)_A = (\mathrm{rev}_A)^*.$$

3 Function spaces and lifting

3.1 Functional types

With the help of the general notions defined above, we can define function spaces (which in category theory are referred to as exponentials, cf. [8]). We characterize function spaces by a functional type, the evaluation operator and the Curry-operator which is used to partially parameterize functions.

Definition 4 Let A be a type. For every type C, the function space from A to C is defined by

- a type

  $$C \leftarrow A$$

- a function

  $$\cdot_C : (C \leftarrow A) \times A \to C$$

  which applies a function to an argument.

- for every function $\oplus : (B \times A) \to C$ a unique function

  $$\widehat{\oplus} : B \to (C \leftarrow A)$$

  (sometimes written as $\oplus\widehat{\ }$) such that

  $$\cdot_C \circ (\widehat{\oplus} \times \mathrm{i}_A) = \oplus. \qquad \text{(A7)}$$

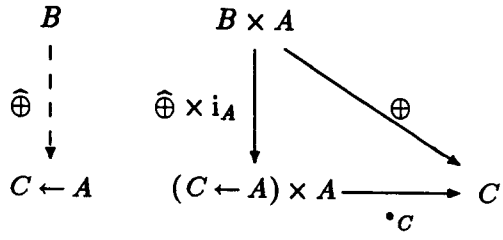

Figure 5: Function space

This is shown in the commuting diagram of figure 5. The dashed line again indicates that $\widehat{\oplus}$ is the *unique* function such that the diagram commutes.

□

It is well-known (from category theory) that this definition completely characterizes function spaces.

One can convince oneself that $C \leftarrow A$ really is the type of functions from A to C by looking at its structure. In order to do so, we construct constants of type $C \leftarrow A$. If we substitute $\mathbf{1}$ for B in the above definition (remember that $A \cong \mathbf{1} \times A$), we obtain for every function

$$f : A \to C$$

a (unique) function

$$\widehat{f} : \mathbf{1} \to (C \leftarrow A).$$

Now $\widehat{f}$ is a constant, and we write

$$\ulcorner f \urcorner = \widehat{f} : \mathbf{1} \to (C \leftarrow A)$$

which is called the *name* of f. In fact, there is a one-to-one correspondence between f and $\ulcorner f \urcorner$, and therefore we may identify the type $C \leftarrow A$ with all the functions from A to C (in the category of sets).

We can name functions with higher arity, like $\oplus : B \times A \to C$ as well, but now there is an important difference between

$$\widehat{\oplus} : B \to (C \leftarrow A)$$

and

$$\ulcorner \oplus \urcorner : \mathbf{1} \to (C \leftarrow (B \times A))$$

where the latter is the name of $\oplus$.

The function $\widehat{\oplus}$ is the curried version of $\oplus$, as can be shown by instantiating A7 with constants $a : \mathbf{1} \to A$ and $b : \mathbf{1} \to B$:

$$\begin{array}{lll} \oplus \circ \langle b\,,\, a\rangle & = & \text{(A7)} \\ {}^{\bullet}_{C} \circ (\hat{\oplus} \times \mathrm{i}_A) \circ \langle b\,,\, a\rangle & = & \text{(D4, A3)} \\ {}^{\bullet}_{C} \circ \langle \hat{\oplus} \circ b\,,\, a\rangle. & & \end{array}$$

After section 3.2 we shall write this as:

$$b \oplus a = (\hat{\oplus} b)a. \tag{D5}$$

Various other forms can be obtained by instantiating the commuting diagram of the definition with operators of different arity. The Curry-operator allows us to partially parameterize functions. To obtain the name of a function, we must 'curry away' all its parameters. It is also possible to uncurry an operator $\otimes : B \to (C \leftarrow A)$, giving

$$\check{\otimes} : B \times A \to C, \quad \check{\otimes} = {}^{\bullet}_{C} \circ (\otimes \times \mathrm{i}_A).$$

It follows immediately from the definition of currying (A7) that

$$\check{\hat{\oplus}} = \oplus = \hat{\check{\oplus}}.$$

By repeating the above derivation for the name of a function, we obtain an interesting connection between composition and application (see figure 6):

$$\begin{array}{lll} {}^{\bullet}_{C} \circ ({}^{\ulcorner}f^{\urcorner} \times \mathrm{i}_A) \circ a & = & \text{(D2, D4, A3)} \\ {}^{\bullet}_{C} \circ ({}^{\ulcorner}f^{\urcorner} \times a) & = & \text{(A7)} \\ f \circ a & & \end{array}$$

which we shall write later as:

$${}^{\ulcorner}f^{\urcorner} \bullet a = f \circ a. \tag{D6}$$

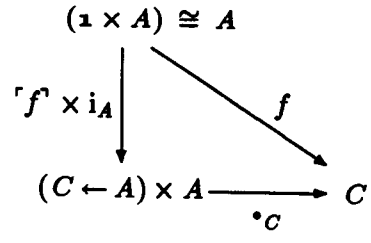

Figure 6: Applying the name of a function

3.2 A notational convention

It is conventional in functional programming not to write the composition operator in ground expressions (i.e. expressions with $\mathbb{1}$ as their domain):

$$f \circ x \quad \text{becomes} \quad fx$$

for $x : \mathbb{1} \to A$. Likewise,

$$\oplus \circ \langle x\,,\, y \rangle \quad \text{becomes} \quad x \oplus y$$

(see figure 7). We shall use these conventions to state the results of our derivations in an

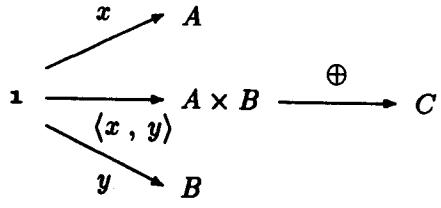

Figure 7: Notational convention

easily readable way.

Also, functional programmers rarely distinguish between a function

$$A \xrightarrow{f} C$$

and its name

$$\mathbb{1} \xrightarrow{\ulcorner f \urcorner} (C \leftarrow A).$$

If we identify these, we have

$$f \bullet_C a = \ulcorner f \urcorner \bullet_C a$$

and by D6

$$\ulcorner f \urcorner \bullet_C a = f \circ a = fa$$

so application becomes invisible, just like composition.

We shall avoid this notational convention in our proofs, because it obscures the derivations. We shall use it in the results of the derivations, however, because it allows us to state these in a way which is customary in functional programming.

3.3 The lifting-laws

Definition 4 has a special structure (called *adjunction* in category theory) which enables us to derive the following properties (e.g. [8, p.128]).

Proposition 1 There is a functor $- \leftarrow A$ for every type A, which maps a type B to $B \leftarrow A$, the type of functions from A to B. The image of a function $f : B \to C$ under this functor is

$$f \leftarrow \mathrm{i}_A : (B \leftarrow A) \to (C \leftarrow A)$$

which is defined as

$$f \leftarrow \mathrm{i}_A = (f \circ {}^\bullet_B)\widehat{\ }. \tag{A8}$$

In a later section it will become clear why we write $f \leftarrow \mathrm{i}_A$ instead of $f \leftarrow A$ (analogous to $f \times \mathrm{i}_A$).

Proof The action on functions preserves identity and composition:

- $$\mathrm{i}_B \leftarrow \mathrm{i}_A \quad = \quad \text{(A8)}$$

  $$(\mathrm{i}_B \circ {}^\bullet_B)\widehat{\ } \quad = \quad \text{(identity)}$$

  $$\widehat{{}^\bullet_B}.$$

  From A7 it follows that $\widehat{{}^\bullet_B}$ is the unique function satisfying

  $${}^\bullet_B \circ (\widehat{{}^\bullet_B} \times \mathrm{i}_A) = {}^\bullet_B$$

  and since $\mathrm{i}_{B \leftarrow A}$ satisfies the same condition,

  $$\mathrm{i}_B \leftarrow \mathrm{i}_A = \mathrm{i}_{B \leftarrow A}.$$

- We know that (A8)

  $$(g \circ f) \leftarrow \mathrm{i}_A = (g \circ f \circ {}^\bullet_B)\widehat{\ }$$

  is the unique function satisfying (A7)

  $$g \circ f \circ {}^\bullet_B = {}^\bullet_D \circ (((g \circ f) \leftarrow \mathrm{i}_A) \times \mathrm{i}_A).$$

  Also,

  $$g \circ f \circ {}^\bullet_B \quad = \quad \text{(A7)}$$

  $$g \circ {}^\bullet_C \circ ((f \circ {}^\bullet_B)\widehat{\ } \times \mathrm{i}_A) \quad = \quad \text{(A7)}$$

  $${}^\bullet_D \circ ((g \circ {}^\bullet_C)\widehat{\ } \times \mathrm{i}_A) \circ ((f \circ {}^\bullet_B)\widehat{\ } \times \mathrm{i}_A) \quad = \quad \text{(A3, A8)}$$

  $${}^\bullet_D \circ (((g \leftarrow \mathrm{i}_A) \circ (f \leftarrow \mathrm{i}_A)) \times \mathrm{i}_A)$$

and therefore

$$(g \leftarrow \mathrm{i}_A) \circ (f \leftarrow \mathrm{i}_A) = (g \circ f) \leftarrow \mathrm{i}_A.$$

□

Proposition 2 The function

$$\bullet_C : (C \leftarrow A) \times A \to C$$

is natural in C, i.e. it is part of a natural transformation

$$\bullet : (- \leftarrow A) \times A \xrightarrow{\cdot} \mathrm{I}.$$

Proof We check the natural transformation property A4 (see the lower part of figure 8):

$$\begin{array}{lll} f \circ \bullet_B & = & \text{(A7)} \\ \bullet_C \circ ((f \circ \bullet_B)\hat{} \times \mathrm{i}_A) & = & \text{(A8)} \\ \bullet_C \circ ((f \leftarrow \mathrm{i}_A) \times \mathrm{i}_A) & & \end{array} \qquad \text{(D7)}$$

□

If we combine this with A7, we find

$$\begin{array}{lll} f \circ g & = & \text{(A7 on } g\text{)} \\ f \circ \bullet_B \circ (\ulcorner g \urcorner \times \mathrm{i}_A) & = & \text{(D7)} \\ \bullet_C \circ ((f \leftarrow \mathrm{i}_A) \times \mathrm{i}_A) \circ (\ulcorner g \urcorner \times \mathrm{i}_A) & = & \text{(product composition)} \\ \bullet_C \circ (((f \leftarrow \mathrm{i}_A) \circ \ulcorner g \urcorner) \times \mathrm{i}_A). & & \end{array}$$

Because $\ulcorner f \circ g \urcorner$ is the *unique* function satisfying

$$f \circ g = \bullet_C \circ (\ulcorner f \circ g \urcorner \times \mathrm{i}_A) \quad \text{(A7 on } f \circ g\text{)}$$

we conclude that

$$(f \leftarrow \mathrm{i}_A) \circ \ulcorner g \urcorner = \ulcorner f \circ g \urcorner \qquad \text{(D8)}$$

(see figure 8). We are not really interested in the type A in $f \leftarrow \mathrm{i}_A$, and therefore we use f° as a shorthand, whenever the type A is clear from the context. This conforms to the notation for the *lifting* operator used in functional programming. Using the notational conventions, D8 becomes:

$$(f^\circ)g = f \circ g.$$

This is the well-known lifting law for unary functions [1], and we therefore say that the function space-constructor $- \leftarrow A$ (denoted $-^\circ$ or $-^\circ$ on functions) is the lifting-functor.

We can rewrite the functor property (A3) for $- \leftarrow A$ to

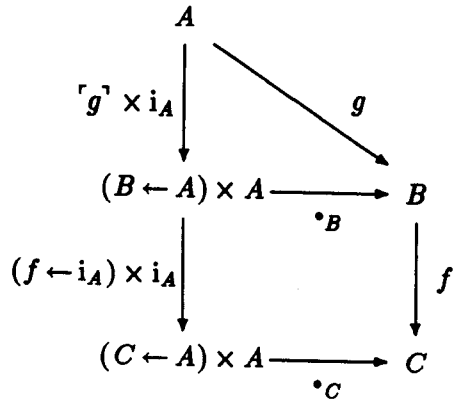

Figure 8: Lifting a unary fuction

$$(f \circ g)^\circ = f^\circ \circ g^\circ \tag{D9}$$

which shows that lifting distributes over composition. If we interpret lifting as curried composition, we might say that composition distributes over itself.

Before we try to lift constants, we note that $(\mathbf{1} \leftarrow A) \cong \mathbf{1}$, since there is only one function from any type to the one-point set. This implies

$$\begin{aligned} &\bullet_{\mathbf{1}} = !_A : A \to ! \\ &!_A \circ a = \mathrm{i}_{\mathbf{1}} \quad \text{for } a : \mathbf{1} \to A. \end{aligned} \tag{D10}$$

The lifting-law for a constant $c : \mathbf{1} \to C$ now becomes

$$\begin{aligned} \bullet_C \circ (c^\circ \times \mathrm{i}_A) &= \quad \text{(D7)} \\ c \circ \bullet_{\mathbf{1}} &= \quad \text{(D10)} \\ c \circ !_A & \end{aligned} \tag{D11}$$

Using the notational convention and instantiating with a A-constant a, we get

$$\begin{aligned} c^\circ \bullet a &= \quad \text{(D10)} \\ c \circ !_A \circ a &= \quad \text{(D10)} \\ c & \end{aligned}$$

which tells us that c° 'eats' its argument and yields c. It appears that $-^\circ$ on constants is the K-combinator.

3.4 Lifting binary operators

It can be proved categorically that functors like $-^\circ$ are *continuous*. Continuity implies that $-^\circ$ preserves products:

$$(C \times B)^\circ \cong C^\circ \times B^\circ \qquad \text{(A9)}\ .$$

In programming languages, this isomorphism is not always implicit, and we then have to explicitly perform a *coercion*:

$$\begin{aligned} \mathrm{co}_1 &= \langle \pi_{C,B}{}^\circ\ ,\ \pi'_{C,B}{}^\circ \rangle : ((C \times B) \leftarrow A) \rightarrow ((C \leftarrow A) \times (B \leftarrow A)) \\ \mathrm{co}_2 &= \langle {}^\bullet C \circ (\pi_{C \leftarrow A, B \leftarrow A} \times \mathrm{i}_A)\ ,\ {}^\bullet C \circ (\pi'_{C \leftarrow A, B \leftarrow A} \times \mathrm{i}_A) \rangle \hat{} \\ &: ((C \leftarrow A) \times (B \leftarrow A)) \rightarrow ((C \times B) \leftarrow A) \end{aligned}$$

Probably, the reader will find the λ-form of these expressions more convenient:

$$\mathrm{co}_1 = \lambda f :: \langle \pi \circ f\ ,\ \pi' \circ f \rangle$$

$$\mathrm{co}_2 = \lambda \langle f\ ,\ g \rangle :: (\lambda x :: \langle fx\ ,\ gx \rangle).$$

It is easy to verify that these are the correct coercions, and that they are inverses of each other.

We shall also need the fact that

$$\ulcorner \langle f\ ,\ g \rangle \urcorner = \mathrm{co}_2 \circ \langle \ulcorner f \urcorner\ ,\ \ulcorner g \urcorner \rangle \qquad \text{(D12)}$$

Proof

$$\begin{aligned} {}^\bullet C,B \circ ((\mathrm{co}_2 \circ \langle \ulcorner f \urcorner\ ,\ \ulcorner g \urcorner \rangle) \times \mathrm{i}_A) &= && (\times \text{ is a functor}) \\ {}^\bullet \circ (\mathrm{co}_2 \times \mathrm{i}_A) \circ (\langle \ulcorner f \urcorner\ ,\ \ulcorner g \urcorner \rangle \times \mathrm{i}_A) &= && (\text{A7 on definition of } \mathrm{co}_2) \\ \langle {}^\bullet \circ (\pi \times \mathrm{i}_A)\ ,\ {}^\bullet \circ (\pi' \times \mathrm{i}_A) \rangle \circ (\langle \ulcorner f \urcorner\ ,\ \ulcorner g \urcorner \rangle \times \mathrm{i}_A) &= && (\text{product properties}) \\ \langle {}^\bullet \circ (\ulcorner f \urcorner \times \mathrm{i}_A)\ ,\ {}^\bullet \circ (\ulcorner g \urcorner \times \mathrm{i}_A) \rangle &= && (\text{A7}) \\ \langle f\ ,\ g \rangle \end{aligned}$$

Since $\ulcorner \langle f\ ,\ g \rangle \urcorner$ is the unique function satisfying

$${}^\bullet C \circ (\ulcorner \langle f\ ,\ g \rangle \urcorner \times \mathrm{i}_A)$$

the two functions are equal. □

We can now derive the lifting law for a binary operator $\oplus : B \times D \rightarrow C$. By A7 and D7 we have the commuting diagram of figure 9. The lifting law follows from:

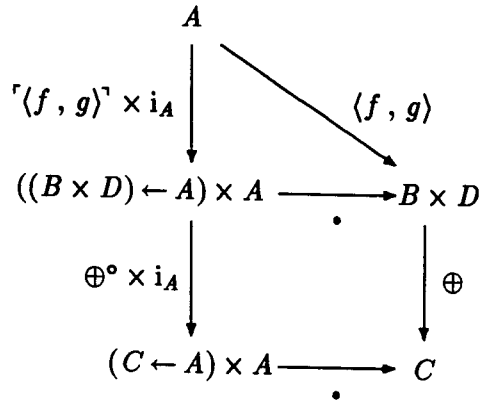

Figure 9: Lifting a binary operator

$$
\begin{array}{lcll}
\oplus \circ \langle f\,,\, g\rangle & = & \text{(A7, D7, figure 9)} & \\
\bullet \circ (\oplus^\circ \times \mathrm{i}_A) \circ (\ulcorner\langle f\,,\, g\rangle\urcorner \times \mathrm{i}_A) & = & \text{(D12, } \times \text{ is a functor)} & \\
\bullet \circ (\oplus^\circ \circ \mathrm{co}_2 \times \mathrm{i}_A) \circ (\langle \ulcorner f\urcorner\,,\, \ulcorner g\urcorner\rangle \times \mathrm{i}_A) & = & \text{(leaving out the coercion)} & \text{(D13)} \\
 & & \text{(notational convention)} & \\
\bullet \circ ((f \oplus^\circ g) \times \mathrm{i}_A). & & &
\end{array}
$$

A more familiar form of this equation is obtained by instantiating with an A-constant a:

$$(fa) \oplus (ga) = (f \oplus^\circ g)a$$

which is the usual definition of lifting for binary operators.

The above derivations can be repeated for operators of any arity, and thus we get the definitions for all lifting operators. We conclude that all lifting definitions, as they appear for instance in [1], are consequences of the single fact that application is a natural transformation.

3.5 Applications

Because we can derive the original definitions, we can also derive other laws on lifted operators. But of course it is much nicer to derive these categorically, so let's try a few. For instance, we can make a generalization of the lifting law for constants: Because there is only one morphism $!_A : A \to \mathbf{1}$ for every type A, we have

$$!_B \circ f = !_A \quad \text{where} \quad f : A \to B.$$

Also, for a constant $c : \mathbf{1} \to C$, we have

$$
\begin{array}{lll}
{}^{\bullet}c \circ ((c \leftarrow \mathrm{i}_A) \times \mathrm{i}_A) & = & \text{(D7)} \\
c \circ {}^{\bullet}_{1} & = & \text{(D10)} \\
c \circ !_A &&
\end{array}
$$

and likewise with B instead of A, so the diagram of figure 10 commutes:

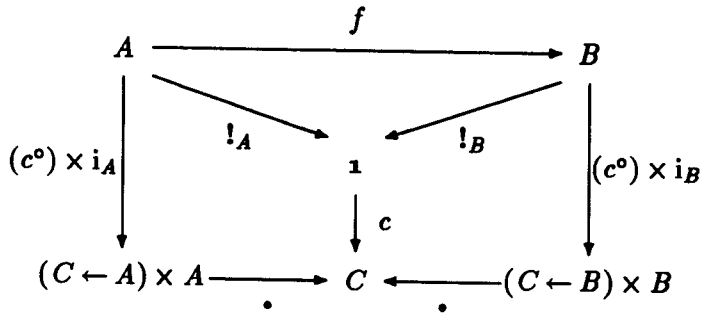

Figure 10: $\check{c^{\circ}}$ is a left-zero of composition

$$^{\bullet} \circ ((c^{\circ}) \times \mathrm{i}_A) = {}^{\bullet} \circ ((c^{\circ}) \times \mathrm{i}_B) \circ f.$$

Using the definition of uncurrying we obtain:

$$\check{c^{\circ}} = \check{c^{\circ}} \circ f \qquad \text{(D14)}$$

If we ignore the types, we might say that $\check{c^{\circ}}$ is a left-zero of composition.

The following example is also taken from [1]. By connecting the appropriate commuting diagrams, we obtain the following (see figure 11):

$$^{\bullet} \circ ((\ulcorner f \urcorner \oplus^{\circ} \ulcorner g \urcorner) \times \mathrm{i}_A) \circ h = {}^{\bullet} \circ ((\ulcorner f \circ h \urcorner \oplus^{\circ} \ulcorner g \circ h \urcorner) \times \mathrm{i}_E).$$

By the definition of uncurrying, this becomes

$$(\ulcorner f \urcorner \oplus^{\circ} \ulcorner g \urcorner)^{\check{}} \circ h = (\ulcorner f \circ h \urcorner \oplus^{\circ} \ulcorner g \circ h \urcorner)^{\check{}}$$

or, using the notational convention and dropping uncurrying

$$(f \oplus^{\circ} g) \circ h = (f \circ h) \oplus^{\circ} (g \circ h). \qquad \text{(D15)}$$

This law tells us that composition distributes over lifted binary operators. Note that we have been rather sloppy about types and uncurrying to obtain the final result.

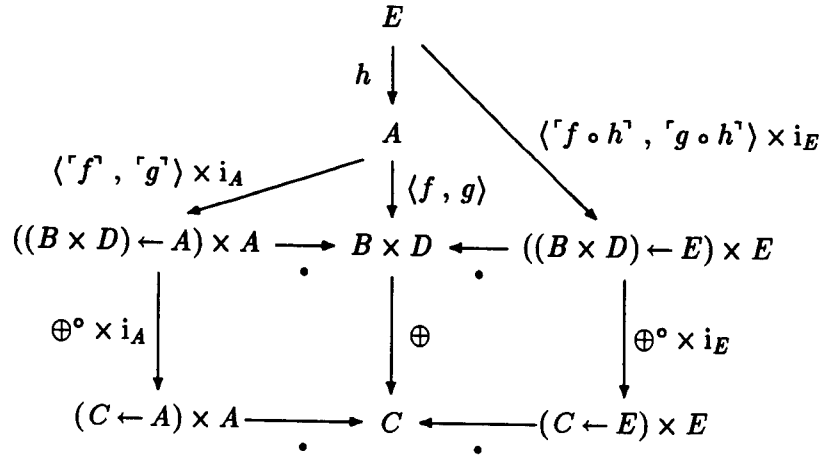

Figure 11: $\circ$ distributes over $\oplus^\circ$

4 Functors and polymorphism

4.1 Strong functors

Consider a function

$$f : B \to C$$

and its name

$$\ulcorner f \urcorner : 1 \to (C \leftarrow B).$$

As we have seen earlier, type constructors are functors, and therefore they can be applied to functions as well as types. For instance the list functor $*$ acts as the 'map' operator on functions, and we have

$$f^* : B^* \to C^*$$

with its name

$$\ulcorner f^* \urcorner : 1 \to (C^* \leftarrow B^*).$$

Similarly, the functional part of the function-space constructor $- \leftarrow A$ is the 'lifting' operator, and we have

$$\ulcorner f^\circ \urcorner : 1 \to ((C \leftarrow A) \leftarrow (B \leftarrow A)).$$

In general, for every functor $\dagger$, we have

$$f^\dagger : B^\dagger \to C^\dagger$$

with its name

$$\ulcorner f^\dagger \urcorner : 1 \to (C^\dagger \leftarrow B^\dagger).$$

We see that the functional part of a functor acts very much like a polymorphic function, for we can repeat the above for any function $f : B \to C$; The functor $\dagger$ maps any function from B to C into a function from $B^\dagger$ to $C^\dagger$, regardless of the types B and C.

In a previous section we argued that polymorphic functions are natural transformations, but now it seems that there is also 'functor-polymorphism'. This problem may be resolved if we can identify the functional part of a functor with a natural transformation. Then the 'functor polymorphism' is just naturality, and we have only one characterization of polymorphism. The natural transformation corresponding to the functional part of the functor $\dagger$ will be called 'dag'.

Definition 5 A functor $\dagger$ is *strong* if there exists a function

$$\mathrm{dag}_{C,B} : (C \leftarrow B) \to (C^\dagger \leftarrow B^\dagger)$$

such that for every function $f : B \to C$

$$\mathrm{dag}_{C,B} \circ \ulcorner f \urcorner = \ulcorner f^\dagger \urcorner. \tag{A10}$$

□

Strong functors are exactly those functors for which a function algorithm exists which computes $\ulcorner f^\dagger \urcorner$ for all functions $\ulcorner f \urcorner$.

4.2 Strong functors and naturality

Proposition 3 The function dag defined in definition A10,

$$\mathrm{dag}_{C,B} : (C \leftarrow B) \to (C^\dagger \leftarrow B^\dagger)$$

is an instance of a natural transformation

$$\mathrm{dag} : (- \leftarrow -) \dot{\longrightarrow} (-^\dagger \leftarrow -^\dagger)$$

The proof constitutes the rest of this section.

The general form of the natural transformation property for $\tau : \dagger \dot{\longrightarrow} \dagger$ is A4:

$$\tau \circ (f^\dagger) = (f^\dagger) \circ \tau.$$

There is a problem however, because the functor $- \leftarrow -$ is *contravariant* in its second argument. This means that if we have

$$g : A \to B \quad \text{and} \quad h : C \to D$$

then

$$(h \leftarrow g) : (C \leftarrow B) \to (D \leftarrow A) \quad (!!).$$

This function is defined as

$$(h \leftarrow g) = (h \circ \bullet_C \circ (\mathrm{i}_{C \leftarrow B} \times g))\urcorner. \tag{A11}$$

It is easy to verify that

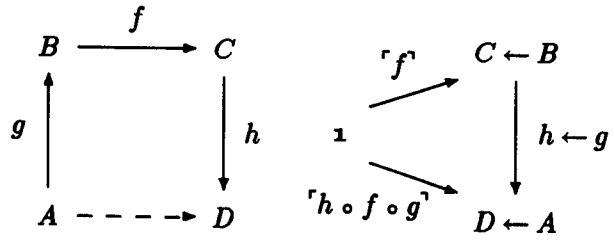

Figure 12: $h \leftarrow g$

$$(h \leftarrow g) \circ \ulcorner f \urcorner = \ulcorner h \circ f \circ g \urcorner \quad \text{(see figure 12).} \tag{D16}$$

This is consistent with A8; if we replace g by i_B we obtain

$$h^\circ = (h \leftarrow \mathrm{i}_B) = (h \circ \bullet_C)\urcorner.$$

We must now check that the natural transformation property holds for dag, i.e. the following holds (see figure 13):

$$\mathrm{dag}_{D,A} \circ (h \leftarrow g) = (h^\dagger \leftarrow g^\dagger) \circ \mathrm{dag}_{C,B}. \tag{D17}$$

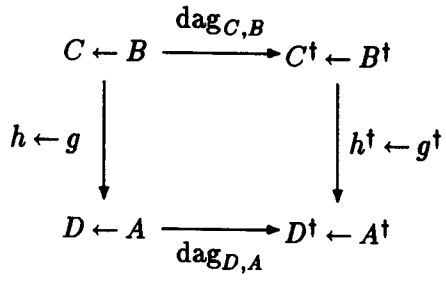

Figure 13: Naturality of dag.

This can be verified as follows:

$$
\begin{aligned}
&\mathrm{dag}_{D,A} \circ (h \leftarrow g) \circ \ulcorner f \urcorner && = && \text{(D16)} \\
&\mathrm{dag}_{D,A} \circ \ulcorner h \circ f \circ g \urcorner && = && \text{(A10)} \\
&\ulcorner (h \circ f \circ g)^\dagger \urcorner && = && \text{(A3)} \\
&\ulcorner (h^\dagger \circ f^\dagger \circ g^\dagger) \urcorner && = && \text{(D16)} \\
&(h^\dagger \leftarrow g^\dagger) \circ \ulcorner f^\dagger \urcorner && = && \text{(A3)} \\
&(h^\dagger \leftarrow g^\dagger) \circ \mathrm{dag}_{C,B} \circ \ulcorner f \urcorner .
\end{aligned}
$$

We may conclude that $\mathrm{dag}_{D,A} \circ (h \leftarrow g) = (h^\dagger \leftarrow g^\dagger) \circ \mathrm{dag}_{C,B}$ if our model satisfies the *extensionality* property, i.e. *for every constant* $x : 1 \to A$, $f \circ x = g \circ x$ *implies* $f = g$. This is true if we work in a category where every type has *enough constants (no junk)*, like the category of sets, which we took as a model. We shall, however, not formally prove this now.

It turns out that extensionality is not necessary for D17 to hold. In [7] a proof is given which does not use extensionality, but this requires some categorical notions which we did not explain in this paper.

We have now proved that to the functional part of a strong functor corresponds a natural transformation. This means that we can still say that polymorphic functions are natural transformations, provided that all functors are strong. Strongness is a very important property of functors used in programming languages, because it means that 'map-like' functions like $f^\dagger$ are computable by applying a higher order function dag to $\ulcorner f \urcorner$.

4.3 Representable functors

The following definition is adapted from [5], and slightly changed, insofar as we use $- \leftarrow R$ instead of the set of all functions from R to a certain type. We can safely do this, since in our model these are isomorphic, as we proved in section 3.1.

Definition 6 A functor $\dagger$ is *representable* if there exist a type R and natural transformations

$$\mathrm{rep} : \dagger \xrightarrow{\cdot} (- \leftarrow R) \quad \text{and} \quad \mathrm{rep}^{-1} : (- \leftarrow R) \xrightarrow{\cdot} \dagger$$

such that

$$\mathrm{rep} \circ \mathrm{rep}^{-1} = \mathrm{i} \leftarrow \mathrm{i}_R \quad \text{and} \quad \mathrm{rep}^{-1} \circ \mathrm{rep} = \mathrm{i}^\dagger$$

(rep is a *natural isomorphism*). We say that rep is a *representation* of $\dagger$, and that R is its representing type. □

Proposition 4 If a functor is representable, then it is strong.

Proof Let

$$\mathrm{rep} : \dagger \xrightarrow{\cdot} (- \leftarrow R)$$

be a representation of $\dagger$. By the definition of representation and A8,

$$\ulcorner f\dagger \urcorner = \ulcorner f \leftarrow \mathrm{i}_R \urcorner = \ulcorner \widehat{f \circ \bullet_B} \urcorner$$

and A10 becomes

$$\mathrm{dag} \circ \ulcorner f \urcorner = \ulcorner (\widehat{f \circ \bullet_B}) \urcorner = \ulcorner (\widehat{\ulcorner f \urcorner \circ \bullet_B}) \urcorner .$$

The *functional completeness theorem* ([4, p.59]) can be used to abstract $\ulcorner f \urcorner$ from this expression, and we obtain:

$$\mathrm{dag}_{C,B} : (C \leftarrow B) \rightarrow ((C \leftarrow R) \leftarrow (B \leftarrow R))$$

$$\mathrm{dag}_{C,B} = \left(\bullet_C \circ \langle \pi_{C \leftarrow B, B \leftarrow R} \circ \pi_{(C \leftarrow B) \times (B \leftarrow R), R} \, , \, \bullet_B \circ (\pi_{C \leftarrow B, B \leftarrow R} \times \mathrm{i}_R) \rangle \right)\widehat{}\,\widehat{}$$

□

Examples of representable functors are:

$- \leftarrow A$: A representing type is A itself, and a representation is

$$\mathrm{i} \leftarrow \mathrm{i}_A : (- \leftarrow A) \xrightarrow{\cdot} (- \leftarrow A)$$

I: The identity functor is represented by

$$\widehat{\pi_{-,1}} : I \xrightarrow{\cdot} (- \leftarrow 1)$$

which follows from the isomorphism between $A \leftarrow 1$ and A.

$- \times -$: The cartesian product is represented by $(1, 1)$, and

$$\pi, \pi' : A \times B \xrightarrow{\cdot} (A \leftarrow 1), (B \leftarrow 1)$$

with its inverse $\langle - \, , \, - \rangle$.

∞: Infinite lists or streams over a type A are represented by functions from the natural numbers N to A:

$$\mathrm{th} : \infty \xrightarrow{\cdot} (- \leftarrow N).$$

Thanks to the above proposition, the natural transformation corresponding to these functors is readily available.

4.4 Abstract data types

Abstract data types can be implemented categorically by *Hagino-functors* [6, 10]. It is of course desirable that such functors are strong, so that their action on functions is a natural transformation. It is, however, beyond the scope of this paper to give a full treatment of this subject. In a forthcoming paper, we prove that Hagino-functors are indeed strong [11].