Categorical Semantics as a Basis for Program Transformation

Nico Verwer

Technical Report RUU-CS-90-38
December 1990

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

Categorical Semantics as a Basis for Program Transformation

Nico Verwer

RUU-CS-90-38
December 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

Categorical Semantics as a Basis for Program Transformation

Nico Verwer

Abstract

Program transformation in the Bird-Meertens formalism is based on so-called laws, which are equalities between expressions. It is shown how these laws can be derived in a very general way, starting from equations which hold in a particular semantics, namely cartesian closed categories.

We argue that datatypes correspond to *strong* functors, which means that their action on functions can be expressed as a program. By proving that functors defined by initial algebras are strong, we establish the validity of our approach for deriving laws about abstract data types. We also derive very general formulations of the factorization and promotion laws for homomorphisms.

1 Introduction

Since the beginning of the seventies, formal semantics has been used to define programming languages in a precise and unambiguous way. In the Scott-Strachey approach, semantics is a mapping form the syntactic domain, the phrases in the language itself, to some semantic domain, which is a rigourously defined, well understood mathematical object. Scott's domain theory provides a mathematical object which is rich and powerful enough to serve as a semantic domain.

In principle, the formal semantics of a language could be used as a guide for constructing compilers and interpreters, and as a reference for programmers. In practice, however, it is rarely used as such. This is partly because the formal semantics of a 'real world' programming language tends to become very large, and a great part of it is merely a list of tedious clauses which state things that every computer scientist takes for granted.

In this paper, we intend to show that formal semantics can serve yet another purpose; it can be used to derive laws which hold for programming constructions. These laws can be used to transform an expression in the language into a *semantically equivalent* expression. This implies that a semantics is needed to check the equivalence of expressions. The other way round, semantics could be used as a guide, to find out exactly which expressions are equivalent, as well.

Why would one want to transform an expression into a semantically equivalent expression?

For many years, software design has been a difficult task. The problem of program size has partially been solved by the introduction of such constructs as modules, packages and polymorphic functions, but it is still impossible to take in the meaning of a big program at a glance. This is because program texts are full of necessary details which divert ones mind from the essence, the intention of the program.

To alleviate this problem, various program specification techniques have been developed. The idea is that one starts with an abstract specification of a program, which specifies *what* the program is supposed to solve, but not *how*. Thus one leaves out all details like specific data structures, algorithms, etc. A specification does not even have to be executable. Implementation is the process of filling in the details, chosing suitable data structures, efficient algorithms, etc. This is the task of the programmer.

The specification is a phrase in some specification language. It has a certain semantics. The final implementation is a phrase in (probably) a different implementation language. It is considered to be correct if it has the same meaning as the specification, i.e. it is semantically equivalent. This semantic equivalence is not necessarily an equality; it is rather an inclusion. For instance, the specification may be nondeterministic, so that different implementations which are not mutually semantically equivalent may be considered correct implementations.

The task of the programmer is to bridge the gap between the specification and an executable, efficient implementation. This is done by stepwise refinement of the specification; in every step, some detail is filled in, and a new program, somewhere between specification and implementation, is obtained. Of course, the programmer must assert that the result of every step is semantically equivalent to the specification. It is unfeasable to determine the exact meaning of intermediate programs, using the semantics of the language in which they are written, and compare this to the semantics of the specification. Therefore, techniques have been developed to *transform* programs into semantically equivalent (but, hopefully, more efficient, executable) programs. Thus the correctness of every step is guaranteed.

The *Bird-Meertens formalism* (BMF) [1, 4, 11, 16], also called *Squigol* because of the squiggly symbols it uses, is a powerful method for transforming programs. Some algorithms have been derived in a neat and systematic way, within this formalism, using simple but powerful transformation rules, or laws. However, the number of laws has become very large, and we feel that some unification and systemization of laws would be advantageous.

This paper shows how many programming laws in BMF can be derived rather than postulated. Spivey has derived laws that hold for lists in a categorical framework [15]. We try to take a more abstract approach, and to derive more general programming laws. In order to be able to do this, we start from one particular model (semantic domain) of functional programming languages, namely Cartesian Closed Categories. The significance of this is, that the laws arise purely from the semantics of our language, and contain no implicit 'extra' assumptions. Also, our laws will be in their 'most general' form, because we do not derive them within the context of some specific application.

The more general the laws are, the more useful they are in (semi-)automatic systems for program transformation. Such systems are simpler to design and use if the fewer the laws they implement. Some of the ideas put forward in this paper have been used in an existing program transformation system [3, 19].

Our contribution is that we show how programming laws can be derived from equalities between arrows in a Cartesian Closed Category, via the internal language of a CCC. Because we make a distinction between the language (BMF) and the model (CCC) it is necessary to prove that type functors are strong. Once we have done this, we can derive some very general programming laws. We also establish the strongness of Hagino functors, which are used to model type constructors. Finally we derive the factorization and promotion theorems of BMF in a categorical framework.

This paper is a revised version of the one that appeared in the proceedings of *Comput-*

ing Science in the Netherlands 90. It clarifies some points and corrects some errors made in the CSN 90 paper.

2 The categorical model

In this section, we define the categorical model that we shall use as the semantic domain in later sections. Since this is standard theory, we review the concept of a Cartesian Closed Category only briefly. In [17] we give an exposition which is more directed towards those who are acquainted with functional programming. A general reference to elementary category theory is [14].

Basically, the semantic domain we shall consider is a Cartesian Closed Category (CCC). It is well known that CCC's are models of typed λ-calculi (see, for instance, [7]), which, in turn, are 'canonical' functional programming languages. In the next section we shall elaborate on this point.

Definition 1 A *category* C consists of

- a class C_o of *objects*, which have no further structure of themselves;
- for every pair of objects $A, B \in \mathsf{C}_o$, a class $\mathsf{C}(A, B)$ of *arrows*. We write $f : A \to B$ for an arrow $f \in \mathsf{C}(A, B)$;
- for every triple $A, B, C \in \mathsf{C}_o$, a *composition* operator $\circ : \mathsf{C}(B, C), \mathsf{C}(A, B) \to \mathsf{C}(A, C)$, which is associative, i.e. $(h \circ g) \circ f = h \circ (g \circ f)$;
- for every object $A \in \mathsf{C}_o$ an arrow id_A, the *identity* on A. For $f : A \to B$ we have $\mathrm{id}_B \circ f = f = f \circ \mathrm{id}_A$.

□

We shall model data types by objects and (typed) functions by arrows.

In order to be useful as a model of λ-calculus, a category must be cartesian closed, i.e. cartesian product and function spaces must exist.

Definition 2 A category C is *cartesian closed* if it has all finite products, i.e.

- there is a special object $\mathbf{1} \in \mathsf{C}_o$, the *terminal object*, such that for every object $A \in \mathsf{C}_o$ there is a unique arrow $!_A : A \to \mathbf{1}$;
- for every pair of objects $A, B \in \mathsf{C}_o$ there is an object $A \times B \in \mathsf{C}_o$, the *cartesian product* of A and B, and arrows $\pi : A \times B \to A, \pi' : A \times B \to B$, the *projections*;
- for every $f : C \to A, g : C \to B$ there is a unique arrow $\langle f , g \rangle : C \to A \times B$ such that

  $$f = \pi \circ \langle f , g \rangle, \quad g = \pi' \circ \langle f , g \rangle;$$

  We shall use the notation

  $$f \times g = \langle f \circ \pi , g \circ \pi' \rangle.$$

and if it is closed, i.e.

- for every $B, B \in \mathsf{C}_o$ there is an an object $C \leftarrow B$, the *exponential* from B to C, and an arrow $\varepsilon : (C \leftarrow B) \times B \rightarrow C$, the *evaluation*;
- for every $f : A \times B \rightarrow C$ there is a unique arrow $\hat{f} : A \rightarrow (C \leftarrow B)$, such that $\varepsilon \circ (\hat{f} \times \mathrm{id}_B) = f$, which is expressed by the following *commuting diagram*:

$$\begin{tikzcd}
A \times B \arrow[d, "\hat{f} \times \mathrm{id}_B"'] \arrow[dr, "f"] & \\
(C \leftarrow B) \times B \arrow[r, "\varepsilon"'] & C
\end{tikzcd}$$

□

We shall use $\langle f , g \rangle$ to model pairing, ε to model function application. The operation $\hat{}$ corresponds to currying of functions. From the uniqueness of $\langle f , g \rangle$ it is easy to prove that $\langle f , g \rangle \circ h = \langle f \circ h , g \circ h \rangle$.

Two objects $A, B \in \mathsf{C}$ are *isomorphic* if there are arrows $f : A \rightarrow B$ and $g : B \rightarrow A$ such that $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$. We then write $f : A \cong B : g$. Examples are unit, commutativity and associativity of the cartesian product:

$$\pi' : 1 \times A \cong A : \langle !_A , \mathrm{id}_A \rangle$$

$$\langle \pi' , \pi \rangle : A \times B \cong B \times A : \langle \pi' , \pi \rangle$$

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

In the category of sets, the terminal object is a one-point set; the unique arrow from a set A to 1 is the function which maps every element to the one point. A constant is a function with no parameters, so the value of the constant may be seen as the image of the one-point under this function. We model this by a *constant arrow* $a : 1 \rightarrow A$, a term which we shall use in any CCC.

We need a way to construct new data types from existing ones. The categorical concept which models a type constructor is a functor. On objects (types), functors operate as type constructors. In addition, functors operate on arrows. This extension of the usual notion of type constructor proves to be very useful, and has, although not explicitly, long been used in functional programming languages.

Definition 3 A functor $\dagger : \mathsf{C} \rightarrow \mathsf{D}$ consists of

- a mapping $\dagger$ from C_o to D_o;
- for every $A, B \in \mathsf{C}_o$ a mapping $\dagger$ from $\mathsf{C}(A, B)$ to $\mathsf{D}(A^\dagger, B^\dagger)$;

and preserves identity and composition:

$$\mathrm{id}_A{}^\dagger = \mathrm{id}_A, \quad (g \circ f)^\dagger = g^\dagger \circ f^\dagger.$$

Composition of functors is forward, i.e. $A^{(\dagger\ddagger)} = (A^\dagger)^\ddagger$. □

We have already seen one functor, namely the cartesian product $\times$. Other functors are the identity I

$$A^{\mathrm{I}} = A, \quad f^{\mathrm{I}} = f$$

(note the use of the superscript-notation for functor application) and the 'left' functor $\ll$

$$A \ll B = A, \quad f \ll g = f$$

(we shall write $A \ll f$ instead of $\mathrm{id}_A \ll f$). Since all our functors go from C to C (the are so-called *endofunctors)*, we just write $\dagger$ instead of $\dagger : \mathsf{C} \to \mathsf{C}$. For the binary functor $\times$ the above properties become:

$$\mathrm{id}_A \times \mathrm{id}_B = \mathrm{id}_{A \times B}$$
$$(f \circ f') \times (g \circ g') = (f \times g) \circ (f' \times g').$$

The *opposite* of a category C is C^{op}. It is the same as C, except that $\mathsf{C}^{op}(A,B) = \mathsf{C}(B,A)$ (all arrows are reversed).

Definition 4 If we define

$$\begin{array}{lll} C \leftarrow B & & \text{as in definition 2} \\ & & \text{for } B, C \in \mathsf{C} \\ h \leftarrow g & = & (h \circ \varepsilon \circ (\mathrm{id}_{C \leftarrow B} \times g))^\frown : (C \leftarrow B) \to (D \leftarrow A) \\ & & \text{for } g : A \to B, h : C \to D \end{array}$$

then $\leftarrow : \mathsf{C} \times \mathsf{C}^{op} \to \mathsf{C}$ is a functor. (It is *contravariant* in its second argument.) □

Definition 5 Let $\dagger : \mathsf{C} \to \mathsf{D}$ and $\ddagger : \mathsf{C} \to \mathsf{D}$ be functors. A *natural transformation* $\tau : \dagger \underset{\cdot}{\longrightarrow} \ddagger$ assigns to every object $A \in \mathsf{C}$ an arrow

$$\tau_A : A^\dagger \to A^\ddagger,$$

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

$$\tau_B \circ f^\dagger = f^\ddagger \circ \tau_A$$

(this is illustrated in the following commuting diagram).

$$\begin{tikzcd} A^\dagger \arrow[r, "\tau_A"] \arrow[d, "f^\dagger"'] & A^\ddagger \arrow[d, "f^\ddagger"] \\ B^\dagger \arrow[r, "\tau_B"'] & B^\ddagger \end{tikzcd}$$

□

Composition of natural transformations is defined as

$$(\sigma \circ \tau)_A = \sigma_A \circ \tau_A.$$

Functors operate on arrows in the same way as they operate on natural transformations; Let $\tau : \dagger \underset{\cdot}{\longrightarrow} \ddagger$, then

$$\tau^{\ddagger} : \dagger\ddagger \underset{\cdot}{\longrightarrow} \ddagger\ddagger, \quad (\tau^{\ddagger})_A = (\tau_A)^{\ddagger}.$$

An example of a natural transformation is the identity

$$\mathrm{id} : \mathrm{I} \underset{\cdot}{\longrightarrow} \mathrm{I}.$$

3 Categorical semantics and strong functors

The language used in the Bird-Meertens formalism is, essentially, first-order polymorphic typed λ-calculus. It uses a highly streamlined syntax, which makes it suitable for program transformation by calculation [11]. For the moment, however, we are not interested in notational issues; our aim is to derive programming laws from the underlying semantics, which can later be transformed into an appropriate notation.

We shall use cartesian closed categories as semantic domains. It is well-known that a first-order typed λ-calculus with cartesian products is equivalent to a CCC, in the sense that there is a functor L from the category of CCC's to the category of typed λ-calculi, and a functor C in the opposite direction, such that L and C are inverses to eachother, up to isomorphism [7]. The functor C gives a categorical *semantics* of a λ-calculus. The functor L gives the *internal language* of a CCC.

Our aim is to derive laws which hold in the BMF language, starting from equalities which hold in CCC's. Contrary to what one might expect, we do not use the semantic functor C in order to do this, but the functor L. This functor translates (objects and arrows in) a CCC to (types and expressions in) the internal language. In this way, two arrows f and g, which are equal in a CCC, are translated into the law $\bar{L}(f) = \bar{L}(g)$, where $\bar{L}$ is the construction of terms out of arrows used by L. The fact that L and C are inverses (up to isomorphism) guarantees that the laws derived by this method are sound w.r.t. the interpretation of λ-calculi in CCC's.

The internal language is defined by the action of L on objects and arrows [7] (actually, this is the object-part of L):

Definition 6 The objects in a category C become the types in $L(\mathsf{C})$; 1 becomes the one-point type `Unit`, $A \times B$ becomes the type $A \times B$ of pairs of A and B, and $B \leftarrow A$ becomes $A \rightarrow B$, the type of functions from A to B. There may be other, non-structured objects, which become the basic types.

Terms of type A in $L(\mathsf{C})$ are obtained from arrows in $\mathsf{C}(1, A)$, via a mapping $\bar{L}$. These terms may contain variables, which are the images under $\bar{L}$ of a specified set of *indeterminate arrows*. If $x : 1 \rightarrow A$ is an indeterminate arrow, $\bar{L}(x : 1 \rightarrow A)$ is the variable $x : A$.

A constant arrow $a : 1 \rightarrow A$, which is not a compound, is mapped to a basic constant $a : A$. The term $\bar{L}(\mathrm{id}_1 : 1 \rightarrow 1)$ is the one-point `unit` : `Unit`.

On compositions, $\bar{L}$ is defined as $\bar{L}(f \circ e) = f(\bar{L}(e))$, where $f : A \to B$ is not a composition of other arrows, and $e : 1 \to A$. For arrows $a : 1 \to A$, $b : 1 \to B$, we define $\bar{L}(\langle a , b \rangle) = (\bar{L}(a), \bar{L}(b))$. Application ε becomes application in BMF (which is written as juxtaposition), projections become projections in BMF, etcetera.

Next, we should define $\bar{L}$ on curried arrows. If we look at definition 2, we see that the only way to obtain an arrow $\widehat{h} : 1 \to (B \leftarrow A)$ is to curry an arrow $h : 1 \times A \to B$. For any $f : A \to B$, we can make such an arrow, if we put $h = f \circ \pi'_{1,A}$. The arrow $(f \circ \pi'_{1,A})\widehat{\ }$, which we shall write as $\ulcorner f \urcorner$, is called the *name*, or the *internalization* of f. This construction is used to make λ-abstractions in $L(\mathsf{C})$:

$$
\begin{array}{ll}
\bar{L}(\ulcorner \mathrm{id}_A \urcorner) = \lambda x : A.x & \\
\bar{L}(\ulcorner f \urcorner) = f & \text{for a non-compound arrow } f : A \to B \\
\bar{L}(\ulcorner g \circ f \urcorner) = \lambda x : A.\bar{L}(\ulcorner g \urcorner)(\bar{L}(\ulcorner f \urcorner)x) & \text{for } f : A \to B, g : B \to C \\
\bar{L}(\ulcorner \langle f , g \rangle \urcorner) = \lambda x : A.(\bar{L}(\ulcorner f \urcorner)x, \bar{L}(\ulcorner g \urcorner)x) & \text{for } f : A \to B, g : A \to C
\end{array}
$$

etcetera. If our language has a composition operator (.), we may write the term in the right hand side of the third line as $\bar{L}(\ulcorner g \urcorner).\bar{L}(\ulcorner f \urcorner)$. □

Datatype constructors in BMF are modeled by functors. For instance, the type $A \times B$ of pairs of A and B corresponds to the object $A \times B$, which is obtained by applying the functor $- \times -$ to the objects A and B. As we saw in the previous section, a functor operates both on objects (types) and arrows (functions). In most programming languages, little or no attention is paid to the *functorial* character of a datatype constructor. Experience with BMF, however, shows that the part of a datatype constructor that operates on functions is very useful indeed. For instance, the list constructor may be applied to a type A, giving $A*$ (lists with elements of type A), but also to a function $\mathtt{f} : A \to B$, giving $\mathtt{f}* : A* \to B*$, which maps $\mathtt{f}$ to every element of its argument. In BMF, there is a function $* : (A \to B) \to (A* \to B*)$ for every A, B (the *map* function in ML) which maps functions on lists.

With definition 6 in mind, we must have an arrow **map** : $(B \leftarrow A) \to (B^* \leftarrow A^*)$ belonging to the functor $*$, which behaves like the part of $*$ which operates on arrows. The existence of such an arrow means that the function $*$, which is defined as $\bar{L}(\ulcorner \mathbf{map} \urcorner)$ can be expressed in our language. Generalizing to an arbitrary functor $\dagger$, we would like to have a 'mapping' arrow, corresponding to the operation of $\dagger$ on functions. Functors for which there is such an arrow are called strong:

Definition 7 A endofunctor $\dagger : \mathsf{C} \to \mathsf{C}$ is *strong* if the part that operates on arrows has an *internalization*. This internalization assigns to every $A, B \in \mathsf{C}$ an arrow

$$\mathbf{dag}_{A,B} : (B \leftarrow A) \to (B^\dagger \leftarrow A^\dagger)$$

such that for every arrow $f : A \to B$

$$\mathbf{dag}_{A,B} \circ \ulcorner f \urcorner = \ulcorner f^\dagger \urcorner .$$

□

Proposition 8 The functor $\times$ is strong.
Proof: The internalization is

$$\begin{aligned} \mathbf{prod}_{A,B,C,D} &= \langle \varepsilon \circ \langle \pi \circ \pi , \pi \circ \pi' \rangle , \varepsilon \circ \langle \pi' \circ \pi , \pi' \circ \pi' \rangle\rangle^{\wedge} \\ &: (B \leftarrow A) \times (D \leftarrow C) \rightarrow ((B \times D) \leftarrow (A \times C)) \end{aligned}$$

Thus,

$$\mathbf{prod}_{A,B,C,D} \circ \langle \ulcorner f \urcorner , \ulcorner g \urcorner \rangle = \ulcorner f \times g \urcorner .$$

□

Proposition 9 The operation on arrows of the functor $\leftarrow : \mathsf{C} \times \mathsf{C}^{op} \rightarrow \mathsf{C}$ is

$$(h \leftarrow g) \circ \ulcorner f \urcorner = \ulcorner h \circ f \circ g \urcorner$$

for $g : A \rightarrow B$, $f : B \rightarrow C$, $h : C \rightarrow D$.
Proof: By writing out definition 4. □

Proposition 10 The functor $\leftarrow$ is strong.
Proof: The internalization is a function

$$\mathbf{fun}_{A,B,C,D} : (D \leftarrow C) \times (B \leftarrow A) \rightarrow ((D \leftarrow A) \leftarrow (C \leftarrow B))$$

such that

$$\mathbf{fun}_{A,B,C,D} \circ \langle \ulcorner h \urcorner , \ulcorner g \urcorner \rangle = \ulcorner h \leftarrow g \urcorner .$$

A suitable definition is

$$\mathbf{fun}_{A,B,C,D} = (\varepsilon_{D,C} \circ \langle \pi \circ \pi \circ \pi , \varepsilon_{C,B} \circ \langle \pi' \circ \pi , \varepsilon_{B,A} \circ \langle \pi' \circ \pi \circ \pi , \pi' \rangle\rangle\rangle)^{\wedge\wedge}$$

□

The CCC C is *well-pointed* if for all $f, g : A \rightarrow B$, and all constants $a : 1 \rightarrow A$ the *extensionality property* holds:

$$(f \circ a) = (g \circ a) \quad \Rightarrow \quad f = g.$$

Sometimes this is expressed by saying that 1 is a *generator*. We shall use this property in the next proposition.

Proposition 11 The internalization of a strong functor $\dagger$ is a natural transformation

$$\mathbf{dag} : (- \leftarrow -) \xrightarrow{\cdot} (-^{\dagger} \leftarrow -^{\dagger})$$

Proof: We prove that **dag** satisfies the natural transformation property, using extensionality. Let $g : A \rightarrow B$, $h : C \rightarrow D$, $f : B \rightarrow C$, then

$$\begin{array}{ll} \mathbf{dag}_{A,D} \circ (h \leftarrow g) \circ \ulcorner f \urcorner & = \text{(proposition 9)} \\ \mathbf{dag}_{A,D} \circ \ulcorner h \circ f \circ g \urcorner & = \text{(definition 7)} \\ \ulcorner (h \circ f \circ g)^{\dagger} \urcorner & = \text{($\dagger$ preserves composition)} \\ \ulcorner h^{\dagger} \circ f^{\dagger} \circ g^{\dagger} \urcorner & = \text{(proposition 9)} \\ (h^{\dagger} \leftarrow g^{\dagger}) \circ \ulcorner f^{\dagger} \urcorner & = \text{(definition 7)} \\ (h^{\dagger} \leftarrow g^{\dagger}) \circ \mathbf{dag}_{B,C} \circ \ulcorner f \urcorner . & \end{array}$$

If our category is well-pointed, we may drop $\ulcorner f \urcorner$ at the end, and we have the natural transformation property left. □

In quite a different context, Martí-Oliet and Meseguer [10] have proved the same, without using extensionality. Their proof is much more complicated, but it shows that we do really not have to require well-pointedness.

Why are we interested in natural transformations? Several authors [2, 13, 14] have remarked that natural transformations can be used to model polymorphic functions. Although there is no general agreement that this is the right characterization of polymorphism, it conforms to the kind of polymorphism that is present in BMF.

A polymorphic function $\mathtt{f}$ is a family of functions, such that for every type A there is an instance $\mathtt{f}[A]$ of $\mathtt{f}$. Polymorphic functions must not depend on the type with which they are instantiated. For example, the function **reverse** reverses the order of the elements of a list, regardless of their type. This means that we may apply some function $\mathtt{f}$ to every element, without affecting the effect of **reverse**. Then it makes no difference if we apply $\mathtt{f}$ before or after **reverse** is applied to the list. This property can be expressed by the programming law

$$\text{For all } A, B, \mathtt{f} : A \to B, \quad \mathtt{reverse}[B].(\mathtt{f}*) = (\mathtt{f}*).\mathtt{reverse}[A].$$

Since the element type is not important, we say that **reverse** is a polymorphic function from lists to lists, or

$$\mathtt{reverse} : * \underset{\cdot}{\longrightarrow} *.$$

Generalizing this, we say that polymorphic functions are natural transformations from one datatype (functor) to another. For a polymorphic function $\tau : \dagger \underset{\cdot}{\longrightarrow} \ddagger$, the above condition becomes precisely the natural transformation property

$$\tau_B \circ f^{\dagger} = f^{\ddagger} \circ \tau_A \quad (\text{for all } A, B, f : A \to B).$$

Identifying polymorphic functions and natural transformations gives a limited form of polymorphism, since there is no type of 'polymorphic functions from † to ‡'. This would correspond to an object $\ddagger \leftarrow \dagger$, which doesn't exist [1]. All we know is that there is an instance of a polymorphic function for every type. This is much like the kind of polymorphism in ML or, indeed, BMF. The implication for us is, that our laws will be often of the form "for all (types) $A, B, \ldots$"; we need quantification at the metalevel.

4 Program transformation laws

As we have said already in the introduction, we want to derive laws which tell us how to transform a program into a semantically equivalent program. In this section, we shall derive some laws. The derivations start from equalities in the semantic domain, which is a CCC. We believe that this is a fruitful approach, because such equalities arise from the structure underlying the language, and they may not be very obvious syntactically. Most of the work presented here has been described in an earlier report [17].

The natural transformation property (see definition 5) holds for polymorphic functions, as we saw in the previous section. This leads to many useful laws in BMF. In the most general form the law reads:

[1] This does exist in models of second order λ-calculus, but these are much more complicated then CCC's.

Proposition 12 If $\mathbf{g} : \dagger \underset{\cdot}{\longrightarrow} \ddagger$ is a polymorphic function, then for all $\mathbf{f} : A \to B$,

$$\mathbf{g}[B].\mathbf{f}\dagger = \mathbf{f}\ddagger.\mathbf{g}[A].$$

Proof: According to the previous section, $\mathbf{g}$ is modeled by a natural transformation g. That implies for any $f : A \to B$

$$\ulcorner g_B \circ f^{\dagger} \urcorner = \ulcorner f^{\ddagger} \circ g_A \urcorner .$$

Translating this into the internal language gives the required result. □

This proposition is used in [3, 19] to move functions to the left or to the right of an expression, which in particular cases leads to more efficient programs.

Example 13 The function $\# : * \underset{\cdot}{\longrightarrow} (N \lll)$ from lists to natural numbers counts the number of elements in a list. Although we have not proved it, it is a natural transformation, so if we have $f : A \to B$,

$$\#_B \circ f^* \quad = \quad (N \lll f) \circ \#_A \quad = \quad \mathrm{id}_N \circ \#_A \quad = \quad \#_A .$$

This allows us to eliminate f from the expression, to make it more efficient. □

Of course, we need to prove that a function is polymorphic, and therefore may be described by a natural transformation, before we can apply these laws. In many cases we know this, for instance if a function corresponds to the internalization of a strong functor.

From proposition 10, we can determine the internalization of the functor $- \leftarrow A$, which is

$$\mathbf{lift}_{A,B,C} =$$

$$\mathbf{fun}_{A,A,B,C} \circ \langle \mathrm{id}_{C \leftarrow B} \, , \, \ulcorner \mathrm{id}_A \urcorner \circ \,!_{C \leftarrow B} \rangle : (C \leftarrow B) \to ((C \leftarrow A) \leftarrow (B \leftarrow A)).$$

In BMF, we write $\mathbf{h}^{\circ}$ instead of $\mathbf{lift}_{A,B,C}\mathbf{h}$, leaving the types (especially A) implicit. With proposition 9 we derive

$$\varepsilon \circ \langle \mathbf{lift}_{A,B,C} \circ \ulcorner h \urcorner \, , \, \ulcorner f \urcorner \rangle =$$

$$\varepsilon \circ \langle \ulcorner h \leftarrow \mathrm{id}_A \urcorner \, , \, \ulcorner f \urcorner \rangle =$$

$$(h \leftarrow \mathrm{id}_A) \circ \ulcorner f \urcorner =$$

$$\ulcorner h \circ f \urcorner$$

which translated into BMF becomes

$$(\mathbf{h}^{\circ})\mathbf{f} = \mathbf{h}.\mathbf{f},$$

the lifting-law for unary functions, as it appears in e.g. [1].

Since $- \leftarrow A$ is a functor, it preserves composition. This leads to another law,

$$(\mathbf{g}.\mathbf{f})^{\circ} = \mathbf{g}^{\circ}.\mathbf{f}^{\circ}$$

which shows that lifting distributes over composition. If we interpret lifting as the curried form of composition, we might say that composition distributes over itself. The fact that functors preserve composition leads to many other 'distributive' laws like this, which are often encountered in BMF program derivations.

With proposition 9 we can derive lifting laws for functions of any arity. This is done in [17] for constants and binary operators. As an example, we shall do this for constant functions $c : \mathbf{1} \to C$. This is a special case, since

$$c \leftarrow \mathrm{id}_A : (\mathbf{1} \leftarrow A) \to (C \leftarrow A)$$

and there is only one function from A to $\mathbf{1}$, namely $!_A$. Therefore $\ulcorner !_A \urcorner$ is the only possible argument to which $\mathbf{lift}_{A,\mathbf{1},C} \circ c$ can be applied, and we write

$$\begin{aligned} c^{\circ} &= \varepsilon \circ \langle \mathbf{lift} \circ c \,, \ulcorner !_A \urcorner \rangle \\ &= \ulcorner c \circ !_A \urcorner : \mathbf{1} \to (C \leftarrow A). \end{aligned}$$

Applying this to an arrow $a : \mathbf{1} \to A$, we get

$$\varepsilon \circ \langle \ulcorner c \circ !_A \urcorner \,, a \rangle = c \circ !_A \circ a = c$$

which becomes the lifting law for constants

$$\mathsf{c}^{\circ}\mathsf{a} = \mathsf{c}.$$

5 Abstract data types in a CCC

Although BMF started as a theory of programs for list-processing, it was soon realized that it could equally well be used with other data structures, such as trees or matrices [4, 12, 16]. A large part of the theory is about *homomorphisms* on these data structures, and their usage in program transformation. Malcolm showed in [8, 9] how Hagino's categorical data types [5] can be incorporated into BMF, to provide a general theory of homomorphisms on abstract data types.

In this section, we first define initial algebras in a CCC (a full treatment would incorporate final coalgebras as well), introducing a notation which is more concise and less forbidding than the one in [8, 9]. We show that under certain conditions the corresponding functors are strong, so that their arrow-part can be internalized. This means that there are **map**-like functions in BMF corresponding to these functors. Most of the material (except the strongness proof) in this and the next section has been described in [18].

Definition 14 A Hagino type functor $\dagger : \mathsf{C} \to \mathsf{C}$ is determined by its *signature*, which consists of

- a binary *components* functor $\textcircled{\dagger} : \mathsf{C} \times \mathsf{C} \to \mathsf{C}$.
- a *constructors* arrow for every object A:

$$\gamma_A : A^{\dagger} \textcircled{\dagger} A \to A^{\dagger}.$$

The object $A^{\dagger}$ is defined as the least fixed point of the functor $(-\textcircled{\dagger}A)$. This is equivalent to saying that $(A^{\dagger}, \gamma_A)$ is a free initial algebra, or that for every object B and arrow $f : B \textcircled{\dagger} A \to B$ there is a unique arrow $(\!| f |\!) : A^{\dagger} \to B$, which is a *homomorphism*:

$$(\!| f |\!) \circ \gamma_A = f \circ ((\!| f |\!) \textcircled{\dagger} \mathrm{id}_A).$$

$$\begin{tikzcd}
A^{\dagger}\textcircled{$\dagger$}A \arrow[r, "\gamma_A"] \arrow[d, "(\!|f|\!)\textcircled{$\dagger$}\mathrm{id}_A"'] & A^{\dagger} \arrow[d, dotted, "(\!|f|\!)"] \\
B\textcircled{$\dagger$}A \arrow[r, "f"'] & B
\end{tikzcd}$$

If $f : A \to B$ then $f^{\dagger} : A^{\dagger} \to B^{\dagger}$, the $\dagger$-map of f, is defined as

$$f^{\dagger} = (\!|\gamma_B \circ (\mathrm{id}_{B^{\dagger}}\textcircled{$\dagger$}f)|\!).$$

$$\begin{tikzcd}
A^{\dagger}\textcircled{$\dagger$}A \arrow[rr, "\gamma_A"] \arrow[d, "(\!|\gamma_B \circ (\mathrm{id}_{B^{\dagger}}\textcircled{$\dagger$}f)|\!)\textcircled{$\dagger$}\mathrm{id}_A"'] & & A^{\dagger} \arrow[d, dotted, "(\!|\gamma_B \circ (\mathrm{id}_{B^{\dagger}}\textcircled{$\dagger$}f)|\!) = f^{\dagger}"] \\
B^{\dagger}\textcircled{$\dagger$}A \arrow[r, "\mathrm{id}_{B^{\dagger}}\textcircled{$\dagger$}f"'] & B^{\dagger}\textcircled{$\dagger$}B \arrow[r, "\gamma_B"'] & B^{\dagger}
\end{tikzcd}$$

□

We can easily show that this defines a functor, with the help of the following

Proposition 15 (unique extension property).

$$h \circ \gamma_A = f \circ (h\textcircled{$\dagger$}\mathrm{id}_A) \quad \Rightarrow \quad h = (\!|f|\!).$$

Proof: This follows from the unicity of $(\!|f|\!)$, given f, $\textcircled{$\dagger$}$ and A. □

The $\dagger$ defined above is really a functor: With the unique extension property we have

$$\begin{aligned}
\mathrm{id}_{A^{\dagger}} &= (\!|\gamma_A|\!) \\
&= (\!|\gamma_A \circ (\mathrm{id}_{A^{\dagger}}\textcircled{$\dagger$}\mathrm{id}_A)|\!) \\
&= \mathrm{id}_A{}^{\dagger}
\end{aligned}$$

where the last step follows from the definition of $\dagger$. This proves that Hagino functors preserve identities. With the unique extension property, it is easy to show that a Hagino functor also preserves compositions.

The object $A^{\dagger}$ is a fixed point of $-\textcircled{$\dagger$}A$, because the arrow $\gamma_A : A^{\dagger}\textcircled{$\dagger$}A \to A^{\dagger}$ is an isomorphism; its inverse is

$$\gamma_A^{-1} = (\!|\gamma_A\textcircled{$\dagger$}\mathrm{id}_A|\!) : A^{\dagger} \to A^{\dagger}\textcircled{$\dagger$}A.$$

The arrow γ_A^{-1} splits a $A^{\dagger}$-term in its components. It is the 'pattern matching function' which is used (implicitly) in functional programming languages to do case-analysis on the structure of a term. This gives a recursive definition of $(\!|f|\!)$, which follows from the homomorphism property:

$$(\!|f|\!) = f \circ ((\!|f|\!)\textcircled{$\dagger$}\mathrm{id}_A) \circ \gamma_A^{-1}.$$

The fact that $A^\dagger$ is the *least* fixed point follows from the unicity of $(\!|f|\!)$. Of course, a fixed point does not always exist. In [13], in the section on recursive domain equations, a categorical version is given of Smyth and Plotkin's lemma which says that a fixed point exists if C has an initial object and all finite colimits, and if $-\textcircled{$\dagger$}A$ is ω-cocontinuous.

Proposition 16 The constructors γ_A of a Hagino functor are instances of a natural transformation

$$\gamma : \Lambda X.X^\dagger \textcircled{$\dagger$} X \underset{\cdot}{\longrightarrow} \dagger$$

(the Λ notation is used to show where the arguments of the functor go).
Proof: If we combine the two diagrams of definition 14, we get

$$\begin{array}{lcl}
f^\dagger \circ \gamma_A & = & \text{(def. of } \dagger\text{, homomorphism property)} \\
\gamma_B \circ (\mathrm{id}_{B^\dagger} \textcircled{$\dagger$} f) \circ (f^\dagger \textcircled{$\dagger$} \mathrm{id}_A) & = & \text{(functoriality of } \textcircled{$\dagger$}\text{)} \\
\gamma_B \circ (f^\dagger \textcircled{$\dagger$} f) & &
\end{array}$$

for any A, B. □

In order to define categorical data types with more than one constructor, we must introduce disjoint unions. Malcolm [8, 9] does this at the metalevel, but here we shall assume that our CCC has coproducts:

Definition 17 The category C has *coproducts* if

- for every pair of objects $A, B \in \mathsf{C}_o$ there is an object $A + B \in \mathsf{C}_o$, and arrows $\iota : A \to A + B, \iota' : B \to A + B$.
- for every $f : A \to C$ and $g : B \to C$ there is a unique arrow $[f, g] : A + B \to C$, such that

$$[f, g] \circ \iota = f, \quad [f, g] \circ \iota' = g.$$

□

The object $A + B$ models the type of the disjoint union of A and B, and $[f, g]$ is a 'case'-construction. The coproduct is a functor if we define

$$f + g = [\iota \circ f, \iota' \circ g].$$

Example 18 An example of a Hagino type functor is the cons-list type constructor $*$. Its signature is:

- components (functor) $\circledast$, defined as

$$X \circledast A = 1 + (A \times X), \quad g \circledast f = \mathrm{id}_1 + (f \times g)$$

- constructors (arrow) $[\square, {>}\!\!{+}]_A : A^* \circledast A \to A^*$. This defines two constructor functions:

$$\square : 1 \to A^*, \quad {>}\!\!{+} : A \times A^* \to A^*$$

which are sometimes called **nil** and **cons**, respectively.

If we have arrows $b : \mathtt{1} \to B$ and $\oplus : A \times B \to B$, then $\llparenthesis [b\,,\oplus] \rrparenthesis : A^* \to B$ satisfies the homomorphism property (which we have split in two equations):

$$\llparenthesis [b\,,\oplus] \rrparenthesis \circ \square = b$$
$$\llparenthesis [b\,,\oplus] \rrparenthesis \circ (a >\!\!+ x) = a \oplus (\llparenthesis [b\,,\oplus] \rrparenthesis \circ x)$$

so it replaces $\square$ by b and $>\!\!+$ by $\oplus$. In the case of

$$f^* = \llparenthesis [\square\,, >\!\!+] \circ (\mathrm{id}_{\mathtt{1}} + (f \times \mathrm{id}_{B^*})) \rrparenthesis$$

this becomes

$$f^* \circ \square = \square f^* \circ (a >\!\!+ x) = (fa) >\!\!+ (f^* x)$$

which is the usual definition of f^*.

The inverse of $[\square\,, >\!\!+]$ determines how a list is constructed, and, if it is a cons, splits it into its head and tail. We can see this if we apply the recursive definition of $\llparenthesis - \rrparenthesis$ to f^*, giving

$$\begin{aligned} & f^* = [\square\,, >\!\!+] \circ (\mathrm{id}_{\mathtt{1}} + (f \times \mathrm{id}_B)) \circ (\mathrm{id}_{\mathtt{1}} + (\mathrm{id}_A \times f^*)) \circ [\square\,, >\!\!+]^{-1} \\ \Rightarrow \quad & f^* \circ [\square\,, >\!\!+] = [\square\,, >\!\!+ \circ (f \times f^*)] \end{aligned}$$

which is the usual definition again, although in a more concise form. This definition also follows directly from the fact that the list constructors $[\square\,, >\!\!+]$ are a natural transformation

$$[\square\,, >\!\!+] : \Lambda X.\mathtt{1} + (X \times X^*) \underset{\cdot}{\longrightarrow} *.$$

□

If we want to translate the results we obtained for Hagino type functors into laws in the internal language, these functors must be strong (see section 3). We can then derive programming laws in the same way as we did with the laws for lifting. Since the translation of equalities in a CCC into laws in BMF is rather straightforward, we shall only prove that Hagino functors are strong, provided that our category meets certain requirements.

In order to show that a Hagino functor $\dagger$ with signature $(\text{\textcircled{$\dagger$}}, \gamma)$ is strong, we should define a corresponding arrow $\mathbf{dag}_{A,B} : (B \leftarrow A) \to (B^\dagger \leftarrow A^\dagger)$ for all objects A, B, such that

$$\mathbf{dag} \circ \ulcorner f \urcorner = \ulcorner f^\dagger \urcorner.$$

We shall do this in two stages. First we show how to internalize the construction of unique homomorphisms $\llparenthesis - \rrparenthesis$. Then we apply this to the construction of mapping functions from homomorphisms (see definition 14).

This will be a rather complicated construction, so the reader may want to skip the proofs and take the result of propositions 20 and 21 for granted. In the proofs we shall omit most of the subscripts on natural transformations (which indicate their instance type), to improve readability, and because they can be easily inferred from the context.

Definition 19 An object A has fixpoints if for all arrows $f : A \to A$ there exists an arrow $\mathsf{Y}_A f : \mathbf{1} \to A$, such that

$$f \circ \mathsf{Y}_A f = \mathsf{Y}_A f.$$

A category C has all fixpoints, if every $A \in \mathsf{C}$, which has at least one constant arrow $a : \mathbf{1} \to A$ which does not contain Y_A, has fixpoints. □

The condition that there is at least one 'normal' constant arrow avoids inconsistencies such as described in [6]. If we wouldn't require this, we could construct 'constants' like $\mathsf{Y}_A(\mathrm{id}_A)$, which can even be made for empty types! Fixpoints allow us to 'encode' recursive arrows like $\llparenthesis f \rrparenthesis$.

Proposition 20 In a category which has all fixpoints, the homomorphism construction $\llparenthesis - \rrparenthesis$ for a type functor $\dagger$ which has a strong components functor $\textcircled{$\dagger$}$, can be internalized, i.e. for every A, B there is an arrow

$$\mathbf{hmf}_{A,B} : (B \leftarrow (B \textcircled{$\dagger$} A)) \to (B \leftarrow A^{\dagger})$$

such that

$$\mathbf{hmf}_{A,B} \circ \ulcorner f \urcorner = \ulcorner \llparenthesis f \rrparenthesis \urcorner.$$

Proof: We shall define an arrow

$$h : \Big(\big((B \leftarrow A^{\dagger}) \leftarrow (B \leftarrow (B \textcircled{$\dagger$} A))\big) \times (B \leftarrow (B \textcircled{$\dagger$} A))\Big) \times A^{\dagger} \to B$$

such that

$$\mathsf{Y}\widehat{\widehat{h}} : \mathbf{1} \to (B \leftarrow A^{\dagger}) \leftarrow (B \leftarrow (B \textcircled{$\dagger$} A))$$

is the internalization of $\mathbf{hmf}_{A,B}$, which can then be defined as

$$\mathbf{hmf}_{A,B} = \varepsilon \circ \langle \mathsf{Y}\widehat{\widehat{h}} \circ\, ! \,, \mathrm{id} \rangle.$$

The arrow h is the 'encoding' of the recursive definition of $\llparenthesis - \rrparenthesis$:

$$h = \varepsilon \circ \langle \pi' \circ \pi \,, \varepsilon \circ \langle \mathbf{odag} \circ \langle \varepsilon \circ \pi \,, \ulcorner \mathrm{id} \urcorner \circ\, ! \rangle \,, \gamma^{-1} \circ \pi' \rangle \rangle.$$

Here **odag** is the internalization of $\textcircled{$\dagger$}$;

$$\mathbf{odag} \circ \langle \ulcorner f \urcorner \,, \ulcorner g \urcorner \rangle = \ulcorner f \textcircled{$\dagger$} g \urcorner.$$

The following diagram illustrates how h is constructed:

$$\begin{tikzcd}
\Big(\big((B \leftarrow A^{\dagger}) \leftarrow (B \leftarrow (B \textcircled{$\dagger$} A))\big) \times (B \leftarrow (B \textcircled{$\dagger$} A))\Big) \times A^{\dagger} \arrow[d, "\mathrm{id} \times \gamma^{-1}"] \\
\Big(\big((B \leftarrow A^{\dagger}) \leftarrow (B \leftarrow (B \textcircled{$\dagger$} A))\big) \times (B \leftarrow (B \textcircled{$\dagger$} A))\Big) \times (A^{\dagger} \textcircled{$\dagger$} A) \arrow[d, "\langle \pi' \circ \pi \,, \langle \mathbf{odag} \circ \langle \varepsilon \circ \pi \,, \ulcorner \mathrm{id} \urcorner \circ\, ! \rangle \,, \pi' \rangle \rangle"] \\
(B \leftarrow (B \textcircled{$\dagger$} A)) \times (((B \textcircled{$\dagger$} A) \leftarrow (A^{\dagger} \textcircled{$\dagger$} A)) \times (A^{\dagger} \textcircled{$\dagger$} A)) \arrow[d, "\mathrm{id} \times \varepsilon"] \\
(B \leftarrow (B \textcircled{$\dagger$} A)) \times (B \textcircled{$\dagger$} A) \arrow[d, "\varepsilon"] \\
B
\end{tikzcd}$$

□

Proposition 21 **In a category which has all fixpoints, a Hagino functor †, with strong components functor ⓣ, is itself strong.**
Proof: The internalization of † is

$$\begin{aligned}&\mathbf{dag}_{A,B} = \\ &\mathbf{hmf}_{A,B} \circ \Big((\varepsilon_{B,B^\dagger \textcircled{$\dagger$} B} \circ (\ulcorner \gamma_B \urcorner \circ ! \times \mathrm{id}_{B^\dagger \textcircled{$\dagger$} B})) \leftarrow \mathrm{id}_{B \textcircled{$\dagger$} A}\Big) \circ \mathbf{odag} \circ \langle \ulcorner \mathrm{id} \urcorner \circ ! \,, \mathrm{id}\rangle.\end{aligned}$$

If we compose this with an arrow $\ulcorner f \urcorner$, we obtain

$$\begin{aligned}
&\mathbf{dag} \circ \ulcorner f \urcorner && = \text{(definition of } \mathbf{dag}\text{)}\\
&\mathbf{hmf} \circ ((\varepsilon \circ (\ulcorner \gamma \urcorner \circ ! \times \mathrm{id})) \leftarrow \mathrm{id}) \circ \mathbf{odag} \circ \langle \ulcorner \mathrm{id} \urcorner \,, \ulcorner f \urcorner \rangle && = \text{(definition of } \mathbf{odag}\text{)}\\
&\mathbf{hmf} \circ ((\varepsilon \circ (\ulcorner \gamma \urcorner \circ ! \times \mathrm{id})) \leftarrow \mathrm{id}) \circ \ulcorner \mathrm{id} \textcircled{$\dagger$} f \urcorner && = \text{(proposition 9)}\\
&\mathbf{hmf} \circ \ulcorner \varepsilon \circ (\ulcorner \gamma \urcorner \circ ! \times \mathrm{id}) \circ (\mathrm{id} \textcircled{$\dagger$} f) \circ \mathrm{id} \urcorner && = \text{(definition 2)}\\
&\mathbf{hmf} \circ \ulcorner \gamma \circ (\mathrm{id} \textcircled{$\dagger$} f) \urcorner && = \text{(definition of } \mathbf{hmf}\text{)}\\
&\ulcorner (\!| \gamma \circ (\mathrm{id} \textcircled{$\dagger$} f) |\!) \urcorner && = \text{(definition 14)}\\
&\ulcorner f^\dagger \urcorner .
\end{aligned}$$

□

6 Factorization and promotion laws

In this section we show how to derive some of the most important laws in BMF. We shall do this by deriving equations between arrows in CCC's. The translation to BMF is then a trivial exercise, because of proposition 21.

Definition 22 A homomorphism $(\!| f |\!) : A^\dagger \to C$ is *factorable* if its underlying arrow $f : C \textcircled{$\dagger$} A \to C$ can be decomposed into

$$f = \oplus \circ (\mathrm{id}_C \textcircled{$\dagger$} g)$$

for some $\oplus : C \textcircled{$\dagger$} B \to C$ and $g : A \to B$. □

In [18] we compare this notion of factorability with the more restricted one given in [9]. As its name already indicates, a factorable homomorphism can be factored:

Proposition 23 If $(\!| f |\!) : A^\dagger \to C$ is factorable as in the above definition, then

$$(\!| f |\!) = (\!| \oplus |\!) \circ g^\dagger .$$

Proof:

$$\begin{aligned}
&(\!| \oplus |\!) \circ g^\dagger \circ \gamma_A && = \text{(definition 14 on } g^\dagger\text{)}\\
&(\!| \oplus |\!) \circ \gamma_B \circ (\mathrm{id}_{B^\dagger} \textcircled{$\dagger$} g) \circ (g^\dagger \textcircled{$\dagger$} \mathrm{id}_A) && = \text{(definition 14 on } (\!| \oplus |\!)\text{)}\\
&\oplus \circ ((\!| \oplus |\!) \textcircled{$\dagger$} \mathrm{id}_B) \circ (\mathrm{id}_{B^\dagger} \textcircled{$\dagger$} g) \circ (g^\dagger \textcircled{$\dagger$} \mathrm{id}_A) && = \text{(functoriality of ⓣ)}\\
&\oplus \circ (\mathrm{id}_C \textcircled{$\dagger$} g) \circ (((\!| \oplus |\!) \circ g^\dagger) \textcircled{$\dagger$} \mathrm{id}_A) && = \text{(decomposition of } f\text{)}\\
&f \circ (((\!| \oplus |\!) \circ g^\dagger) \textcircled{$\dagger$} \mathrm{id}_A)
\end{aligned}$$

By proposition 15, the proposition holds. □

The following diagram may be helpful in understanding the factorization of homomorphisms.

$$\begin{tikzcd}
A^{\dagger}\obar A \arrow[rr, "\gamma_A"] \arrow[dd, "\llparenthesis f\rrparenthesis\obar \mathrm{id}_A"'] & & A^{\dagger} \arrow[d, "g^{\dagger}"'] \arrow[dd, bend left=60, "\llparenthesis f\rrparenthesis"] \\
 & & B^{\dagger} \arrow[d, "\llparenthesis \oplus \rrparenthesis"'] \\
C\obar A \arrow[r, "\mathrm{id}_C\obar g"] \arrow[rr, bend right=40, "f"'] & C\obar B \arrow[r, "\oplus"] & C
\end{tikzcd}$$

The literature on BMF is full of applications of the factorization of homomorphisms. If $\oplus : C\obar C \to C$, the homomorphism $\llparenthesis \oplus \rrparenthesis$ is called a *reduction* [2], and $\llparenthesis f\rrparenthesis$ can be factored into a map followed by a reduction. For examples, the reader should consult [1, 4, 9, 15, 16] or other publications on program transformation in BMF.

The following proposition is also often used in program transformation. It is called the promotion theorem.

Proposition 24 Let f, g, h be arrows between objects as indicated in the following diagram:

$$\begin{tikzcd}
A^{\dagger}\obar A \arrow[r, "\gamma_A"] \arrow[d, "\llparenthesis f\rrparenthesis\obar \mathrm{id}_A"'] & A^{\dagger} \arrow[d, "\llparenthesis f\rrparenthesis"'] \arrow[dd, bend left=60, "\llparenthesis g\rrparenthesis"] \\
B\obar A \arrow[r, "f"'] \arrow[d, "h\obar \mathrm{id}_A"'] & B \arrow[d, "h"'] \\
C\obar A \arrow[r, "g"'] & C
\end{tikzcd}$$

If h is $f \to g$ promotable, i.e.

$$h \circ f = g \circ (h\obar \mathrm{id}_A)$$

then

$$\llparenthesis g\rrparenthesis = h \circ \llparenthesis f\rrparenthesis.$$

Proof:

$$\begin{aligned}
& h \circ \llparenthesis f\rrparenthesis \circ \gamma_A && = && \text{(definition 14 on } \llparenthesis f\rrparenthesis\text{)} \\
& h \circ f \circ (\llparenthesis f\rrparenthesis\obar \mathrm{id}_A) && = && (h \text{ is } f \to g \text{ promotable}) \\
& g \circ (h\obar \mathrm{id}_A) \circ (\llparenthesis f\rrparenthesis\obar \mathrm{id}_A) && = && \\
& g \circ ((h \circ \llparenthesis f\rrparenthesis)\obar \mathrm{id}_A) && &&
\end{aligned}$$

With proposition 15, the proposition follows. □

[2] In BMF, the reduction $\llparenthesis \oplus \rrparenthesis$ would be written as $\oplus/$

Again, we have no room to give more than one example, but there are many in the references we gave earlier.

Example 25 Let $\phi : -\textcircled{\dagger}A \underset{\cdot}{\longrightarrow} \mathrm{I}$. If we define

$$\llparenthesis \phi \rrparenthesis_B = \llparenthesis \phi_B \rrparenthesis$$

for every object B, then this is a natural transformation $\llparenthesis \phi \rrparenthesis : (A^\dagger \ll) \underset{\cdot}{\longrightarrow} \mathrm{I}$.
Proof: The naturality of ϕ means that any function $h : B \rightarrow C$ is $\phi_B \rightarrow \phi_C$ promotable. According to proposition 24,

$$\llparenthesis \phi \rrparenthesis_C = h \circ \llparenthesis \phi \rrparenthesis_B$$

which is the naturality for $\llparenthesis \phi \rrparenthesis$. □

References

[1] Roland Backhouse, *An exploration of the Bird-Meertens Formalism*, technical report CS8810, dept. of Computing Science, University of Groningen, 1988.

[2] Roland Backhouse, *Making Formality Work For Us*, in *Bulletin of the EATCS, 38*, 1989.

[3] Aswin A. van den Berg, *Attribute Grammar Based Transformation Systems*, vakgroep informatica, rijksuniversiteit te Utrecht, INF/SCR-90-16.

[4] Richard S. Bird, *Lectures on Constructive Functional Programming*, in *Constructive Methods in Computing Science*, NATO ASI F55, Springer-Verlag, 1989.

[5] Tatsuya Hagino, *Category Theoretic Approach to Data Types*, Thesis, University of Edinburgh, 1987.

[6] Hagen Huwig and Axel Poigné, *A note on inconsistencies caused by fixpoints in a cartesian closed category*, Theoretical Computer Science 73 (1990), p. 101–112.

[7] J. Lambek and P.J. Scott, *Introduction to higher order categorical logic*, Cambridge University Press, 1986.

[8] Grant Malcolm, *Homomorphisms and Promotability*, in *Mathematics of Program Construction*, LNCS 375, Springer-Verlag 1989.

[9] Grant Malcolm, *Factoring Homomorphisms*, Technical Report Computing Science Notes CS8909, Dept. of Mathematics and Computing Science, University of Groningen, 1989.

[10] Narciso Martí-Oliet and José Meseguer, *Duality in Closed and Linear Categories*, SRI International, computer science laboratory, SRI-CSL-90-01.

[11] Lambert Meertens, *Algorithmics — Towards Programming as a Mathematical Activity*, in *Proceedings CWI Symposium on Mathematics and Computer Science*, CWI Monographs vol. 1, North-Holland, 1986.

[12] Lambert Meertens, *Firts steps towards the theory of rose trees*, unpublished draft, CWI, Amsterdam, 1988.

[13] Benjamin C. Pierce, *A Taste of Category Theory for Computer Scientists*, technical report CMU-CS-90-113, Carnegie Mellon University (to appear in Computing Surveys).

[14] D.E. Rydeheard and R.M. Burstall, *Computational Category Theory*, Prentice/Hall, 1988.

[15] Mike Spivey, *A Categorical Approach to the Theory of Lists*, LNCS 375, Springer-Verlag, 1989.

[16] STOP (Specification and Transformation Of Programs) project, *Lecture notes, International summer school on constructive algorithmics*, Hollum (Ameland), Utrecht University, 1989.

[17] Nico Verwer, *Deriving Programming Laws Categorically*, technical report RUU-CS-90-6, Utrecht University, dept. of Computer Science, 1990.

[18] Nico Verwer, *Homomorphisms, Factorisation and Promotion*, technical report RUU-CS-90-5, Utrecht University, dept. of Computer Science, 1990.

[19] Harald Vogt, Aswin van den Berg, Arend Freije, *Rapid Development of a Program Transformation System with Attribute Grammars and Dynamic Transformations*, LNCS 461, Springer Verlag, 1990.