# DATA REFINEMENT in a CATEGORICAL SETTING

C. A. R. HOARE. October 1987.

Summary: Programming languages can be regarded as categories; in this view, abstract and concrete data representations are functors between such categories; and the abstraction functions used in data refinement (e.g. VDM) are natural transformations between such functors. A sufficient condition for the validity of data refinement by natural transformation is that the combinators of the programming language are ~~either~~ covariant endofunctors, or natural transformations or adjunctions) (between compositions of such endofunctors.. Many conditionals and cartesian products, as well as more general limits and colimits. Further programming concepts and different kinds of abstraction will be treated in subsequent papers.

programming language concepts satisfy these conditions, for example

## 0. A note on notations

L, M for categories

We shall use the letters
b, c, d for objects  E, F, G, H, for functors
p, q, r, k, l for arrows  n, m, t for natural transformations

We shall use overbars $\overleftarrow{p}$ to denote the source of p, and $\overrightarrow{p}$ the target. We shall equate an object with its identity arrow, even in the case of a graph. Thus the axioms for a graph P can be written

$$\left.\begin{aligned} \overleftarrow{(\overleftarrow{p})} &= \overleftarrow{p} = \overrightarrow{(\overleftarrow{p})} \\ \overrightarrow{(\overrightarrow{p})} &= \overrightarrow{p} = \overleftarrow{(\overrightarrow{p})} \end{aligned}\right\} \text{ for all } p \text{ in } P$$

(These laws help in avoiding double overbars).

Composition in a category will be denoted by semicolon (in diagrammatic order). This will bind more loosely than overbar. Thus the axioms for a category can be written

$p;q$ is defined iff $\overrightarrow{p} = \overleftarrow{q}$

$$\overrightarrow{p;q} = \overrightarrow{q}, \quad \overleftarrow{p;q} = \overleftarrow{p}$$

$$p;\overrightarrow{p} = p = \overleftarrow{p};p$$

$$p;(q;r) = (p;q);r.$$

(These and similar laws help to keep overbars short)

A chain of compositions will normally be written without brackets, and an appeal to associativity will be implicit.

Application of functions will be denoted by juxtaposition. This will bind tighter than semicolon, but looser than overbar.

Thus the defining properties of a Functor F are

$$\overrightarrow{Fp} = F\vec{p}\,, \quad \overleftarrow{Fp} = F\overleftarrow{p}$$

$$F(p;q) = Fp\,;\,Fq$$

and for a natural transformation $n : F \to G$

$$\overleftarrow{nd} = Fd\,, \quad \overrightarrow{nd} = Gd$$

$$Fp\,;\,n\vec{p} = n\overleftarrow{p}\,;\,Gp$$

Juxtaposition will associate to the left, and functions of more than one argument will be usually curried. Thus for example

$$\theta abp = ((\theta a)b)p$$

where (for example) $\theta$ is an adjunction with components $\theta ab$

The reader is expected to be vaguely familiar with the above categorical concepts.

## 1. Introduction.

Data refinement is one of the most effective formal methods for the design and development of large programs and systems. Design starts with a graph $P$, whose nodes are the names $(b, c, .., d)$ of basic types, and whose arrows are the names $(p, q, \ldots, r)$ of primitive operations on values of the type. We will denote the object which is the source of the arrow $p$ by $\overleftarrow{p}$, and its target object by $\vec{p}$.

A program text over graph $P$ in a language $L$ is the text of any syntactically valid and strictly typed program whose primitive types and operations are named by objects and arrows from the graph $P$. These are assembled into a program by means of the combinators of the language $L$. One such combinator is sequential composition, denoted by $(;)$; this and other combinators will be treated in later sections.

An interpretation of a graph P is given by a graph morphism G, mapping each type name (object) of P to a set of mathematical values, and each operation name of P to some mathematical function (or relation, in later parts). The domain (source) of the function Gp is just $G\overleftarrow{p}$ and the codomain is included in $G\overrightarrow{p}$, i.e., the target of Gp. This is because G is a graph morphism, and therefore respects type structure.

A large systems program can now be designed as an abstract program, which applies mathematical functions to sets of mathematical values. In the text of this program, the names b, c, .., d are used to denote the types Gb, Gc, .., Gd; and the names p, q, .., r denote the functions Gp, Gq, .., Gr. Full advantage can be taken of the simplicity and generality of mathematics to ensure correctness of the design.

If the programming language available implements directly all the mathematical concepts used in the abstract program, this can now be compiled and run directly on the computer, and no refinement is necessary. More usually, a general-purpose implementation of mathematical concepts is impossible, or impossibly inefficient. So it is necessary to select or design a specialised implementation of P, taking advantage of the special characteristics of the particular abstract program which uses it. Such a concrete implementation takes the form of another graph morphism F. F maps the type names of P to bitpatterns which can be held in the store of a computer, and the operation names of P to subroutines which manipulate these bit-patterns. The same program text

designed abstractly at the earlier stage is now given this new concrete representation, so that it can be compiled and executed efficiently on a computer.

But of course it is essential to prove that the replacement of the abstract interpretation G by the concrete interpretation F maintains the correctness of the program. In data refinement, this is done with the aid of a collection $n$ of abstraction functions, one for each type-name $d$ of P. The function $nd$ maps the values of the concrete set $Fd$ to the corresponding abstract set $Gd$. The abstraction functions are proved to commute with all the primitive operations $p$ of P, in the following sense

To apply the abstraction function $n\vec{p}$ after a concrete operation $Fp$ gives the same result as applying the abstraction function $n\overleftarrow{p}$ before the corresponding abstract operation $Gp$.

This description presupposes that the sets and functions over which $F$ and $G$ range are included in a homogeneous mathematical space $M$, on which functional or sequential composition (here denoted by semicolon) is defined for type-compatible functions. In other words, $M$ is a category. Thus the commuting principle (described in the inset above) can be expressed in the algebraic laws:

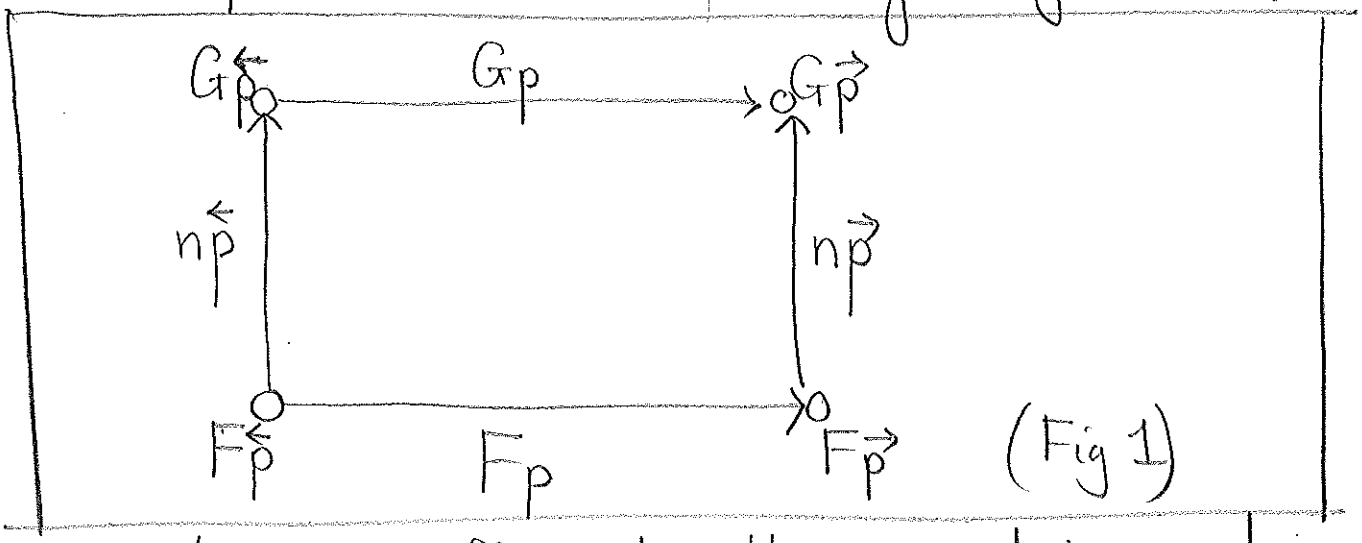
$$Fp ; n\vec{p} = n\overleftarrow{p} ; Gp \quad \text{for all arrows } p \text{ in } P.$$

$$n\overleftarrow{d} = Fd\,, \quad n\vec{d} = Gp, \quad \text{for all objects } d \text{ in } P$$

In category theory, a collection of functions n which satisfy these laws is called a natural transformation from F to G. The laws are often abbreviated to

$$n: F \to G$$

or expanded to a commuting diagram (Fig 1)




(Fig 1)

In data refinement, the commuting equation is proved to hold just for the primitive operations p. in P. It is then believed to hold also when p is allowed to range over all programs in some much more powerful programming language L. The purpose of this paper is to explore the conditions under which such a belief

is valid. These conditions will be expressed as algebraic laws which must be satisfied by the combinators of the programming language L. We will find that the laws are the very laws that define the categorical concepts of a functor, and of a natural transformation, and of an 'adjunction'! Furthermore, many of the useful features of efficiently implementable programming languages satisfy these laws. In later papers, the results of this paper will be generalised to additional features of more powerful languages. Indeed it would seem an important criterion for design of a new programming language that it should maintain the validity of some clearly defined technique of data refinement.

## 2. Composition.

The first programming language combinator which we shall consider is composition (functional or sequential); and to begin with, we assume this to be the only combinator of the language $L$. So the program texts of $L$ constitute the smallest set which contains the names From $P$; and whenever it contains $k$ and $l$ (with $\vec{k} = \overleftarrow{l}$) it also contains $(k;l)$. We assume that the language $L$ obeys the normal laws of programming, namely that composition is associative, and has a left and right unit (identity or SKIP of the appropriate type). Formally

$$j;(k;l) = (j;k);l$$

$$l;\vec{l} = l = \overleftarrow{l};l$$

(here and later, we do not distinguish between an object and its identity). Furthermore,

composition is defined if and only if the target of its first operand is the same as the source of the second operand, and this restriction is enforced by a compile-time check.

These laws are summarised by saying that the language $L$ is a category. In fact, we can define $L$ formally as the free (or path) category $\mathcal{F}P$ over graph $P$ within the category of all categories. This means that (as long as $P$ remains uninterpreted) two programs in $L$ are equal if and only if they can be proved equal by the algebraic laws for a category. But for our purposes, a much more important property of $\mathcal{F}P$ is the following:

Let $H$ be any graph morphism from $P$ to $M$ (or more strictly $UM$, where $U$ is the forgetful functor which maps a category onto its underlying graph). Then there is an unique functor, known as $\phi H$, whose domain is $\mathcal{F}P$, and which agrees with $H$ on $P$, i.e.,

$$\phi H(k;l) = \phi Hk \,;\, \phi Hl \quad \text{for } k, l \in \mathcal{F}P$$

$$\phi H p = Hp \quad \text{for } p \in P$$

The fact that $\phi H$ is uniquely defined by the above equations is guaranteed by the syntax of $L$, which states that every program is a finite nonempty sequence of primitive operations separated by semicolons. In category theory, it is usually expressed by an adjunction diagram

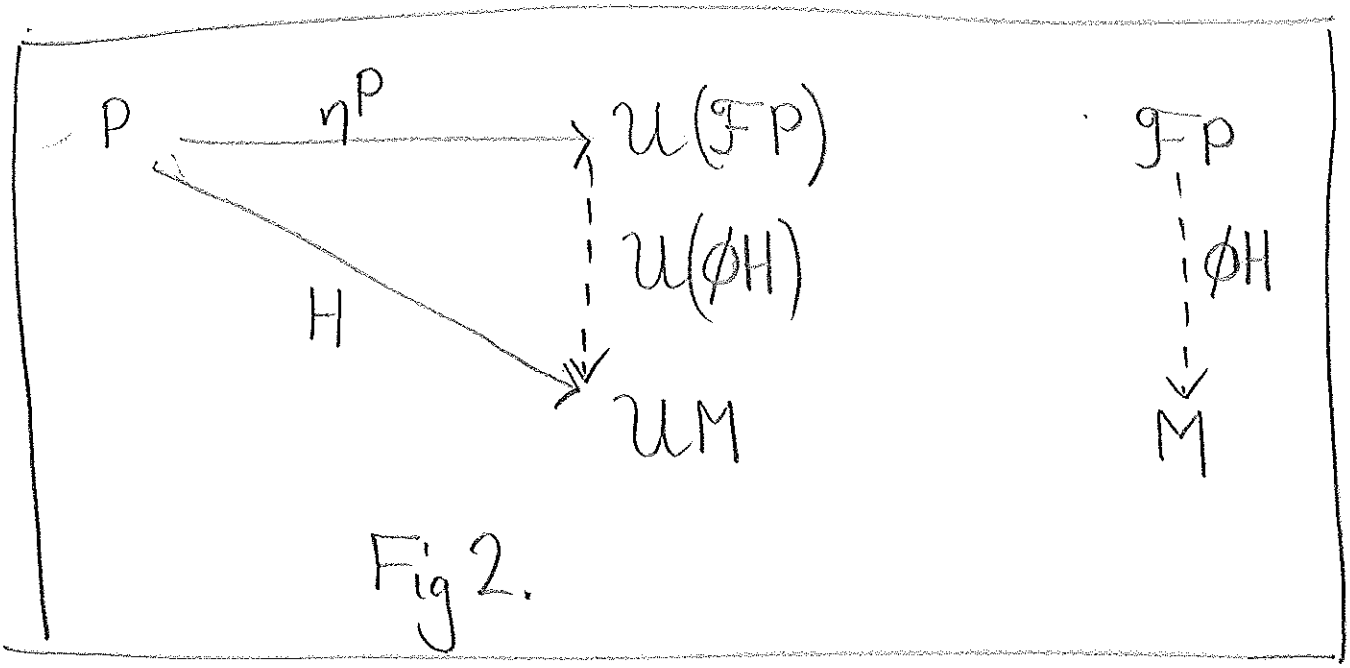



Fig 2.

From the fact that $\phi H$ distributes through semicolon, it follows that for any program text $l$ in $L$, $\phi H l$ is just the function in $M$ which is obtained by interpreting each primitive name $p$ of $P$ within the text $l$ as the mathematical operation $\phi H p$, and taking semicolon to mean functional composition. In other words, $\phi H$ (or indeed, any other functor) provides a semantics for the programming language which is denotational, in the sense that the meaning of the whole is given in terms of the meanings of its parts.

Now we can formulate and prove the important theorem which states that data refinement by natural transformation is a valid development method for programs written in the language L.

Theorem 1.

If $Fp ; n\vec{p} = n\overleftarrow{p} ; Gp$ for all $p$ in $P$
then $\emptyset Fl ; n\vec{l} = n\overleftarrow{l} ; \emptyset Gl$ for all $l$ in $\mathcal{F}P$.

Proof. by induction on the structure of $l$.

Base case: follows from assumption because

$\emptyset Fp = Fp$ and $\emptyset Gp = Gp$ for all $p$ in $P$.

Induction step:

$$\emptyset F(k;l) ; n\overrightarrow{(k;l)}$$

$$\{\emptyset F \text{ is functor, } \overrightarrow{k;l} = \vec{l}\} = (\emptyset Fk) ; (\emptyset Fl) ; n\vec{l}$$

$$\{\text{induction hypothesis}\} = (\emptyset Fk) ; n\overleftarrow{l} ; (\emptyset Gl)$$

$$\{\text{same again, } \vec{k} = \overleftarrow{l}\} = n\overleftarrow{k} ; \emptyset Gk ; \emptyset Gl$$

$$\{\emptyset G \text{ is functor, } \overleftarrow{k;l} = \overleftarrow{l}\} = n(\overleftarrow{k};\overleftarrow{l}) ; \emptyset G(k;l)$$

□

## 3. The category of natural transformations.

In the design of a complex system, the technique of data refinement may be applied in several successive steps. In the first step, the replacement of G by F is justified by a natural transformation

$$n: F \Rightarrow G.$$

Now suppose F in its turn is not adequately concrete or efficient for direct implementation (on a computer). So an even more concrete representation E is designed and justified by

$$m: E \to F.$$

We now wish to be sure that the most concrete Functor E is also a valid implementation of the original abstract design G. This can be guaranteed by finding a natural transformation from E to G. The following theorem shows how this can always be done.

Theorem 2.

If $m: E \to F$ and $n: F \to G$

then $(m;n): E \to G$

where $(m;n)$ is defined as $\lambda d.(md; nd)$

Proof: by calculation. (or see Fig 3).

$El; (m;n)\vec{l}$

$\{$def $(m;n)\}$ $= El; m\vec{l}; n\vec{l}$

$\{m: E \to F\}$ $= m\overleftarrow{l}; Fl; n\vec{l}$

$\{n: F \to G\}$ $= m\overleftarrow{l}; n\overleftarrow{l}; Gl$

$\{$def $(m;n)\}$ $= (m;n)\overleftarrow{l}; Gl$

□

This theorem is not only very useful in the stepwise development of large systems programs; it is also the nub of the proof of a fundamental theorem of category theory, namely that the set of all functors between a category $L$ and a category $M$ are the

objects in a category Nat(L, M), whose arrows are the natural transformations from one such functor to another. An example of such a category is Nat(P, 𝒰M), whose objects are (F, G, ... H) possible interpretations of the graph P in M, and whose arrows are the abstraction functions (n, m, (n;m), ...) which relate these interpretations.

Another example is Nat(𝔽P, M), which contains as objects the extensions ∅F, ∅G, .., ∅H; and (as arrows) the natural transformations between them.

The operator ∅ is just θPM, where θ is the adjunction between the forgetful functor 𝒰 and the free functor 𝔽. It is a bijection from the objects (graph morphisms) of the category Nat(P, 𝒰M) and the objects (functors) of the category Nat(𝔽P, M). In order to ensure the validity of data refinement, we need to know that for every arrow

n: F ⇒ G in Nat(P, 𝒰M)

there exists an arrow $m'$

$$m' : F \to G \quad \text{in} \quad \mathrm{Nat}(\mathbb{F}P, M).$$

This will be true if and only if there exists a function $\phi^+$ from $\mathrm{Nat}(P, \mathbb{U}M)$ to $\mathrm{Nat}(\mathbb{F}P, M)$ which agrees with $\phi$ on objects, and satisfies (fig 4)

$$\phi^+ n : \phi^+ F \to \phi^+ G \qquad \text{for all } n \text{ in } \mathrm{Nat}(P, \mathbb{U}M)$$

Without loss of generality, I think we can insist that $\phi^+$ is a functor, i.e., that it additionally satisfies

$$\phi^+(m;n) = \phi^+ m \,;\, \phi^+ n$$

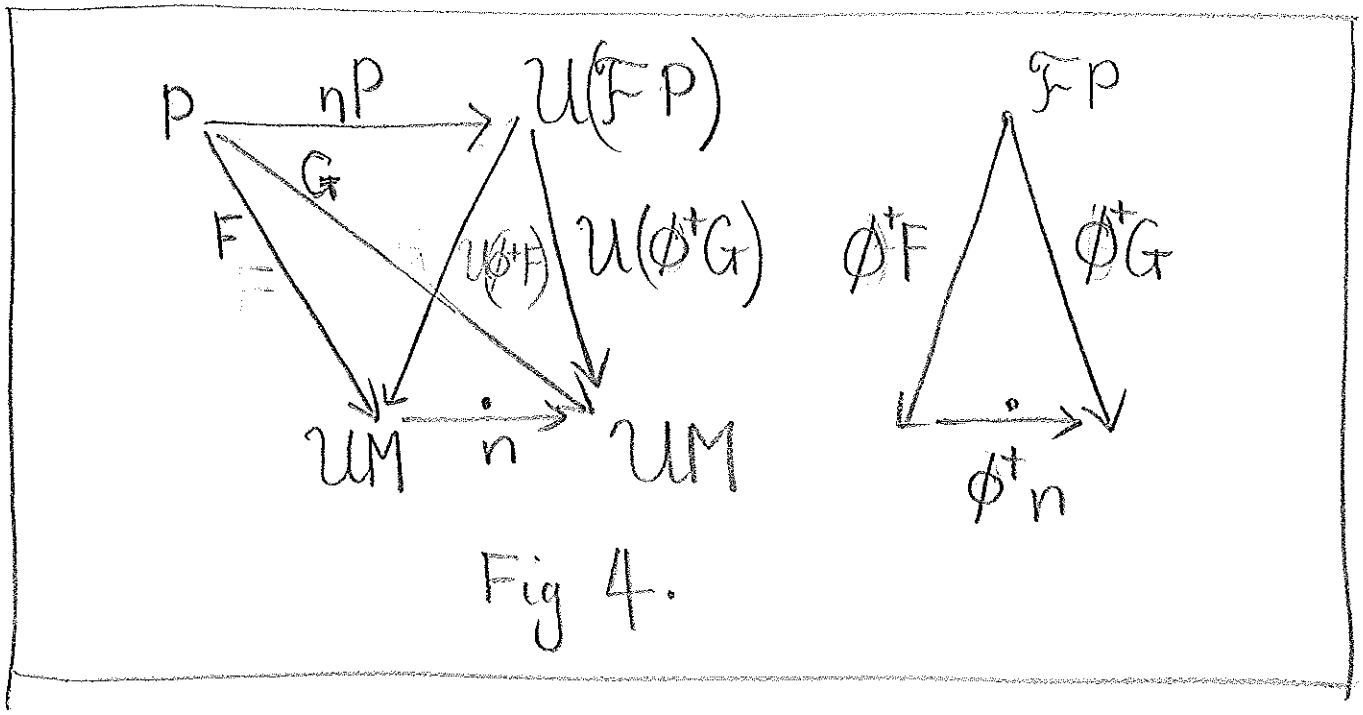



Fig 4.

We therefore formulate the important general definition: an adjunction $\phi$ is said to respects natural transformations if $\phi$PM can be extended to a functor for all P and M.

Now theorem 1. can be reformulated as stating that the operator $\phi$ defined above can be extended to a functor; in fact, in this case, $\phi^+ n$ in M is just the same as n itself in UM. For a more powerful programming language, whose combinators can introduce new data types (as well as new operations on them), the discovery of an extension $\phi^+$ may require invention. But if such an extension exists, then the method of refinement using natural transformations is valid for that language.

Since $\phi$ is a bijection, it has a converse $\check{\phi}$. If this can also be extended to a functor $\check{\phi}^+$ which is the converse of $\phi^+$, then Nat(P, UM) is isomorphic to Nat(FP, M), and natural transformations constitute a complete method of proving refinement in the language. But we shall not pursue the topic of completeness in this paper.

advisable to generalise the descriptions of the previous section to deal with an arbitrary set $\Sigma$ of combinators, one of which is composition. A $\Sigma$-category is defined as a (small) heterogeneous $\Sigma$-algebra, whose sorts are the homsets of a graph P. All combinators in $\Sigma$ (including even composition) are in principle indexed by the homsets of their operands and results. All terms of the algebra can therefore in principle be checked for type consistency. The carrier

A $\Sigma$-morphism between $\Sigma$-algebras L and M is defined as a function from the carrier of L to the carrier of M that commutes (distributes through) all the combinators in $\Sigma$. Since these include composition, every $\Sigma$-morphism is a functor.

insert on page 11

set of a $\Sigma$-category is defined in the usual way as the smallest set closed with respect to syntactically consistent application of the combinators of $\Sigma$.

A $\Sigma$-variety is defined as a category $V$ whose objects are themselves $\Sigma$-categories, and whose arrows are $\Sigma$-morphisms between them. A $\Sigma$-variety is usually defined by a set of equational laws governing the combinators of $\Sigma$; its objects are just those $\Sigma$-categories in which the laws are valid. Since $\Sigma$-morphisms are functors, a $\Sigma$-variety is a subset of the category of small categories.

Following the construction of the previous section, let us fix a $\Sigma$-variety $V$. Let $U$ be the forgetful functor which maps $V$ to the category of small graphs, and let $F$ be its left adjoint, and $\theta$ its adjunction. As before $FP$ serves as the algebraic semantics of the programming language in $V$. The variety $V$ is said to respect natural transformation if for all graphs $P$ and all $M$ in $V$ the adjunction $\theta PM$ be extended to a functor from $Nat(P, UM)$ to $Nat(FP, M)$. When $\Sigma = \{;\}$, and

Using and extending the definition at the end of the previous section, we say that a <u>$\Sigma$-variety $V$ respects natural transformation</u> if the forgetful adjunction between $V$ and the category of graphs is respectful of natural transformation. Let $\Sigma$ be a set of combinators governed by a set of algebraic equations. $\Sigma$ is said to respect natural transformation if the $\Sigma$-variety defined by the equations does so. Loosely, we will say that a collection of combinators respects natural transformation if the addition of such combinators to some previously understood $\Sigma$ continues to respect natural transformation.

add this to my manuscript, please

## 5. The Fundamental theorems.

In the following proofs we will adopt a consistent notation. Let $\mathcal{U}$ be [illegible] with (the forgetful functor $\mathcal{U}$ from a $\Sigma$-variety $\mathbf{V}$ to the category of graphs, ~~and~~ let $\mathcal{F}$ be its left adjoint, and $\theta$ the adjunction from $\mathcal{F}$ to $\mathcal{U}$

Let $n : F \to G$ be a member of $Nat(P, \mathcal{U}M)$.

Define $\phi' = \theta PM$. We now need to extend $\phi$ to $\phi^+$, so that its domain includes all of $Nat(P, \mathcal{U}MP)$, in such a way that we can later prove it to satisfy $\phi^+ n : \phi F \to \phi G$

The definition of $\phi^+ n$ proceeds, in the same way as the definition of $\phi F$ and $\phi G$, by induction on the combinator structure of the elements of the carrier of $\mathcal{F}P$.

Let $P$ be a graph, and $M$ a member of the $\Sigma$-variety $\mathbf{V}$. Let $F$ and $G$ be functors from $P^!$ to $\mathcal{U}M$

## 5.1 Endofunctors.

In this section, we confine attention to signatures containing only combinators. which are governed by the laws which define an endofunctor. Consequently, all objects of $\mathcal{FP}$ are either objects in P, or they are denoted by expressions whose combinators are all endofunctors, and whose ultimate operands denote objects in P

Let p be an object in $\mathcal{FP}$, denoted by an expression S whose combinators are endofunctors and whose (atomic) operands denote elements of P. Since S is an object, it is equal to $\vec{S}$; and since S is composed solely by endofunctors, the superscript distributes through to the atomic operands.

This makes it possible to define a natural transformation $\phi^+ n$ as an extension of $n$, in exactly the same way as a $\Sigma$-morphism $\phi F$ is defined as an extension of a graph morphism $F$.

$$(\phi^+ n)\, b = nb \qquad \text{for objects } b \text{ in } P$$

$$(\phi^+ n)\,(hb) = h(\phi^+ n p) \qquad \text{for } h \in \Sigma,\ b \in \mathcal{F}P$$

Our first task is to prove that $\phi^+$ is a functor, which is achieved by the following pair of lemmas.

Lemma $\phi^+(n;m) = \phi^+ n ; \phi^+ m$

Proof. Induction on the structure of of the object to which both sides are applied.

Base case. b is from P

$\phi^+(n;m)\, b$

$\{\text{def}\,\phi^+, \text{base}\} = (n;m)\, b$

$\{\text{def}\ ;\} \quad = nb ; mb$

$\{\text{def}\,\phi^+\} \quad = \phi^+ n\, b ; \phi^+ m\, b$

$\{\text{def}\ ;\} \quad = (\phi^+ n ; \phi^+ m)\, b$

Induction step

$$
\begin{aligned}
&\phi^+(n;m)(hb) \\
&\{\text{def}\phi^+\} = ' \quad = h(\phi^+(n;m)b) \\
&\{\text{induction}\} \quad = h(\phi^+ n\, b\,;\, \phi^+ m\, b) \\
&\{h \text{ functor}\} \quad = h(\phi^+ n b)\,;\, h(\phi^+ m\, b) \\
&\{\text{def}\phi^+\} \quad = \phi^+ n'(hb);\phi^+ m(hb) \\
&\{\text{def};\} \quad = (\phi^+ n\,;\,\phi^+ m)b
\end{aligned}
$$

□

---

The identities (objects) of the category are established by the lemma. Let $F\Gamma$ be the restriction of the functor $F$ to objects, so that $F\Gamma : F \xrightarrow{\cdot} F$.

Lemma. $(F\Gamma)^+ = (F^+)\Gamma$

Base case: let $b$ be an object of $P$

$$(F\Gamma)^+ b = (F\Gamma)b = Fb = F^+ b = F^+\Gamma$$

Induction step:

$$
\begin{aligned}
&(F\Gamma)^+(fS) \\
&\{\text{def}+\} \quad = F((F\Gamma)^+ S) \\
&\text{induction} \quad = F(F^+ S) \\
&\text{def } F^+ \text{ is } \Sigma\text{-morph} \quad = F^+(FS) \\
&\qquad = ((F^+)\Gamma)(fS)
\end{aligned}
$$

□

The identities of $Nat(P, \mathcal{UM})$ are the restrictions to objects of the graph morphisms from $P$ to $\mathcal{UM}$. If $F$ is such a functor, let $F\restriction$ be its restriction to objects

Lemma $\phi^+(F\restriction) = (\phi F)\restriction$

Proof. Base case: trivial

Induction step:

$(\phi F)\restriction(hb)$

$\{hb \text{ is an object}\} = (\phi F)(hb)$

$\{\phi F \text{ is } \Sigma\text{-morphism}\} = h(\phi F\restriction b)$

$\{\text{induction hypothesis}\} = h(\phi^+(F\restriction) b)$

$\{\text{def } \phi^+\} = \phi^+(F\restriction)(hb)$ □

In future, we shall simplify the notation, writing $n^+$ for $\phi^+ n$ and $F^+$ for $\phi F$. We now come to the simplest case of our fundamental theorem.

Theorem 3: Endofunctors respect natural transformation.

Proof: (let $n : F \to G$ and let $\Sigma$ contain only endofunctors. We use induction on the structure of the carrier set of $\mathcal{F}P$.

Base case. assume $p \in P$, so also $\vec{p} \in P$

$$F^+ p \,;\, n^+ \vec{p}$$

$\{F^+$ extends $F$, $n^+$ extends $n\}$

$$= Fp \,;\, n\vec{p}$$

$\{n : F \to G$, by assumption$\}$

$$= n\vec{p} \,;\, Gp$$

$\{G^+$ extends $G$ and $n^+$ extends $n\}$

$$= n^+ p \,;\, G^+ p$$

~~$n^+(p;q) = (n^+ p);n'$~~

~~$n^+;m^+ \vec{p} = (n;m)^+ \vec{p}$~~

Vd. $n^+ d ; m^+ d = (nd;md)^+$

LHS $n^+(F\vec{p}) ; m^+(F\vec{p})$

$= F(n^+\vec{p} ; m^+\vec{p})$

$= F((n^+ ; m^+)\vec{p})$

$= F(n;m)^+\vec{p}$

$= (n;m)^+(F\vec{p})$

induction step: let $f \in \Sigma$, and consider $f\vec{\ell} \in \mathcal{FP}$

$F^+(f\vec{\ell})\ ;\ n^+(\overrightarrow{f\ell})$

{$F^+$ is $\Sigma$-morphism, and $f \in \Sigma$ is a functor}

$= F(F^+\ell)\ ;\ n^+(f\vec{\ell})$

{definition of $n^+$ for combinator $f$}

$= f(F^+\ell)\ ;\ f(n^+\vec{\ell})$

{$f$ is a functor}

$= f(F^+\ell\ ;\ n^+\vec{\ell})$

{induction hypothesis}

$= f(n^+\overleftarrow{\ell}\ ;\ G^+\ell)$

{by a mirror argument}

$= n^+(\overleftarrow{f\ell})\ ;\ G^+(f\ell)$ □

This argument readily extends to bifunctors or functors of even higher arity, provided that they are covariant.

## 5.2 Natural Transformations

We now extend the range of combinators in $\Sigma$ to include arbitrary endofunctors and also arbitrary natural transformations between them. This addition does not introduce any new objects into $\mathcal{FP}$, and so does not require any extension or change to the definition of $n^+$. We therefore proceed directly to the proof of the main theorem

The next theorem extends the previous result to $\Sigma$ containing both endofunctors and natural transformations between them.

Theorem 4. Natural transformations respect natural transformations.

Proof: let f and g be endofunctors in $\Sigma$ and let t in $\Sigma$ be a natural transformation between them. We need consider additionally those elements of FP which can be expressed in the form td, where d is an object of FP

The main part of the proof is the same as that of Theorem 3

$F^+(td)\ ;\ n^+(\overrightarrow{td})$

$\{td : fd \to gd\}$

$= F^+(td)\ ;\ n^+(gd)$

$\{F^+$ is a $\Sigma$-morphism, and $t \in \Sigma$, def $n^+\}$

$= t(F^+d)\ ;\ g(n^+d)$

{by induction hypothesis: $n^+d : F^+d \to Gd$}

$= t(\overleftarrow{n^+d})\ ;\ g(n^+d)$

$\{t : f \to g$, by assumption$\}$

$= f(n^+d)\ ;\ t(\overrightarrow{n^+d})$

{by a mirror argument}

$= n^+(\overleftarrow{td})\ ;\ G^+(td)$ □

A very similar proof applies to covariant endofunctors of higher arity. The crucial property of such functors is that they admit distribution by $n^+$. (to all their arguments.) So do arbitrary combinations of such endofunctors. So do

certain other functors such as the identity functor, (or constant functors), (or the selector functors of higher arity which select a singlet one of their arguments, e.g.

if $1^{st}\,pq = p$, $2^{nd}\,pq = q$, all $p, q$

then $n^+(1^{st}\,pq) = n^+p = 1^{st}(n^+p)(n^+q)$.

Thus natural transformations between these functors (also respect natural transformations (in any combination)).

$$p;\vec{O}\vec{p}\vec{q} = \vec{O}pq;q$$

In category theory, $Obc$ (if it exists) is called the zero morphism between b and c. In a programming language like $\mathcal{D}[\,]$, it is the abort command. Its inclusion among the combinators of $\Sigma$ is justified by the same proof as Theorem 4.

Setting $p$ to $\overleftarrow{q}$ we deduce

$$\theta_{\overleftarrow{q}\,\overrightarrow{r}}(q;qr) = \theta_{\overleftarrow{q}\,\overleftarrow{r}}\, q\,;r \qquad \text{---} \quad (*)$$

and setting $r$ to $\overrightarrow{q}$ we get

$$\theta_{\overleftarrow{p}\,\overrightarrow{q}}(p;q) = f_p;\theta_{\overrightarrow{p}\,\overrightarrow{q}}\,q \qquad (**)$$

# 5.3. Adjunctions.

If f and g are opposing functors, an adjunction between f and g is an operator $\theta$ of arity three. Its first two operands are objects and its third operand is an arrow. The sorts are constrained by

If $q : b \to gc$

then $\theta bcq : fb \to c$.

The naturality of $\theta bc$ in b and c can be stated as an algebraic law

$$\theta \overleftarrow{p}\, \overrightarrow{r}\, (p ; q ; gr) = fp ; \theta \overrightarrow{p}\, \overleftarrow{r}\, q ; r$$

insert from 36a

... We define an operator which satisfies this law as a junction. A adjunction of the more familiar kind in category theory is just a junction for which $\theta bc$ is a bijection for all b and c. For our purposes, we do not need this stronger property

The following theorem extends the range of of the previous results to junctions.

Theorem 5 Junctions respect natural transformations

Proof: let $f, g, \theta$ be members of $\Sigma$ with the properties described in the preamble to this theorem. The proof is additional to that of Theorem 3, and so considers only elements of the form $\theta bcq$.

$$F^+(\theta bcq);\ n^+(\theta bcq)$$

{F is a $\Sigma$-morphism and $\theta \in \Sigma$; and type constraint on $\theta$}

$$= \theta(F^+b)(F^+c)(F^+q);\ n^+c$$

{by induction, $n^+c : F^+c \to G^+c$, $F^+b = \overleftarrow{F^+}q$, by $*$}

$$= \theta(F^+b)(G^+c)(F^+q\ ;\ g(n^+c))$$

{$g(n^+c) = n^+(gc) = n^+(\overrightarrow{q})$, and (by induction) $n^+ : F^+ \to G^+$}

$$= \theta(F^+b)(G^+c)(n^+\overleftarrow{q}\ ;\ G^+q)$$

{$n^+\overleftarrow{q} : F^+b \to G^+b$ (by induction), so by $**$}

$= f(n^+ b) \,;\, \theta\,(G^+ b)\,(G^+ c)\,(G^+ q)$

{definition of $n^+$, $G^+$ is a $\Sigma$-morphism, $\theta \in \Sigma$}

$= n^+(fb) \,;\, G^+(\theta bcq)$

{type constraint of $\theta$}

$= n^+(\overleftarrow{\theta} bcq) \,;\, G^+(\theta bcq)$ □

This proof readily extends to junctions with two (parallel) arguments, which are governed by the laws:

if $q, q' : b \to gc$

then $\theta bcqq' : fb \to c$

$\theta^{\overleftarrow{p}\,\overrightarrow{r}}\,(p;q;gr)\,(p;q';gr)$

$= fp;\, \theta^{\overrightarrow{p}\,\overleftarrow{r}}\,qq';r$

## 6. Examples.

(1) Suppose we wish to introduce a new object z into the language. This can conveniently be done by including in $\Sigma$ a constant functor K which maps every element to the same new object; this is done by the equation $Kp = Kq$ all $p,q$ in $L$

(2) If we wish to specify $Kp$ to be the terminal object in $\mathcal{FP}$, then for any object $d$ in $\mathcal{FP}$ there is an arrow $!d : d \to K$, which satisfies

$$p ; !\vec{p} = !\overleftarrow{p} ; Kp \qquad \text{for all } p \text{ in } \mathcal{FP}$$

i.e., ! is a natural transformation

$$! : I \to K .$$

If we also postulate the equation

$$!(Kp) = Kp ,$$

then the above equations define ! uniquely, and Kp up to isomorphism. Initial objects can be introduced in a similar fashion. In a category of Scott domains

the terminal object is the domain $\{\perp\}$ containing only the undefined value $\perp$.

(3) The (coproduct) (disjoint union) functor will be denoted by an infix $+$. $b+c$ is the discriminated union type, which appears (for example) in PASCAL as (a variant record). $(p+q)$ is a case discrimination. When applied to a value of type $(\overleftarrow{p}+\overleftarrow{q})$ it first tests which variant it comes from. If it is the first variant, then $p$ is applied, obtaining a result of type $\vec{p}$, which is then injected into the first variant of $(\vec{p}+\vec{q})$. The treatment of the second case is similar. Thus $\overleftarrow{(p+q)} = \overleftarrow{p}+\overleftarrow{q}$ and $\overrightarrow{(p+q)} = \vec{p}+\vec{q}$. Furthermore, it is easy to see that the above description of the case discrimination satisfies the (other) defining property of a bifunctor

$$(p+q);(r+s) = (p;r)+(q;s)$$

There are three natural transformations associated with the coproduct bifunctor, $(\nabla, t, \text{and } f)$.

$$(p+p); \Delta\vec{p} = \Delta\overleftarrow{p}; p$$

$$p; t\vec{p}\vec{q} = t\overleftarrow{p}\overleftarrow{q}; (p+q)$$

$$q; f\vec{p}\vec{q} = f\overleftarrow{p}\overleftarrow{q}; (p+q)$$

$\nabla b : (b+b) \to b$ is the merging operation which forgets which alternative its operand came from; $tbc : b \to (b+c)$ injects its operand as the first variant of the result, and $fbc : c \to (b+c)$ is the injection to the second variant.

These operators provide the means by which a conditional combinator can be introduced into a programming language.

If $e : b \to (\overleftarrow{p} + \overleftarrow{q})$ and $\vec{p} = \vec{q} = c$

then $\underline{\text{if}}\ e\ \underline{\text{then}}\ p\ \underline{\text{else}}\ q : b \to c$

is defined as $e; (p+q); \nabla c$.

Here, the condition is supposed to inject into the first variant if it evaluates to true, and to the second otherwise

(4) A similar treatment can be given to the product bifunctor $p \times q$. The associated natural transformations are the projections $0bc : b \times c \to b$ and $1bc : b \times c \to c$, and the duplicating operation $\Delta b : b \to (b \times b)$, which maps $x$ of type $b$ to the pair $(x, x)$. In a category of total functions, these satisfy

$$p ; \Delta\vec{p} = \Delta\overleftarrow{p} ; p \times p$$

$$(p \times q) ; 0\vec{p}\vec{q} = 0\overleftarrow{p}\overleftarrow{q} ; p$$

$$(p \times q) ; 1\vec{p}\vec{q} = 1\overleftarrow{p}\overleftarrow{q} ; q$$

(5) We denote a zero morphism between objects b and c as $0bc$. The defining property of a zero morphism is

$$p ; 0\vec{p}\vec{q} = 0\overleftarrow{p}\overleftarrow{q} ; q$$

It is therefore a natural transformation between the bifunctors $1^{st}$ and $2^{nd}$. In Dijkstra's language, this property is enjoyed by the abort command, which fails under all circumstances.

(6). Let $(q \sqcap q')$ be the non-deterministic program that behaves either like $q$ or like $q'$, where $q, q' : b \to c$. Composition distributes through non-determinism in both directions:

$$p ; (q \sqcap q') ; r = (p ; q ; r) \sqcap (p ; q' ; r)$$

This merely states that $\sqcap$ is a junction between the identity functor and itself. (more formally, write $q \sqcap q'$ as $0bc\, q\, q'$).

# 7. Counterexamples.

(1) A contravariant functor h is one that satisfies the laws:

$$hp : h\vec{p} \to h\overleftarrow{p}$$

$$h(p;q) = hq ; hp$$

A functor of higher arity may be covariant in some arguments and contravariant in others. Consider for example the exponential bifunctor, denoted by $\Rightarrow$. $(b \Rightarrow c)$ is a function space of functions from b to c. $(p \Rightarrow q)$ is an operation which when applied to a function f delivers (the function) $(p; f; q)$ as result. So type consistency (requires that) if must be in $(\vec{p} \Rightarrow \overleftarrow{q})$ and the result will be in $(\overleftarrow{p} \Rightarrow \vec{q})$. So

$$(p \Rightarrow q) : (\vec{p} \Rightarrow \overleftarrow{q}) \to (\overleftarrow{p} \Rightarrow \vec{q})$$

Furthermore

$(p \Rightarrow q) ; (r \Rightarrow s)$ applied to $f$ is

$$r ; (p ; f ; q) ; s = (r ; p) ; f ; (q ; s)$$

which is the same as $(r ; p) \Rightarrow (q ; s)$ applied to $f$. So we deduce.

$$(p \Rightarrow q) ; (r \Rightarrow s) = (r ; p) \Rightarrow (q ; s) .$$

In summary: $\Rightarrow$ is contravariant in its first operand, covariant in its second.

Contravariant functors, which are of obvious importance in higher order programming languages, do not respect natural transformations. However, they do respect natural isomorphisms and so do the corresponding natural transformations and functions. This will be the subject of a later paper.

(2) In a functional programming language the zero morphisms $0_{bc}$ are identified with $\perp_{bc}$, the partial function (to c) that is undefined on all values of type b. Let $K_z : c \to d$ be the (non-strict) constant function which always gives the result z, even when its operand is undefined. So $K_z(\perp_{bc}(x)) = z$, whereas $\perp_{bd}(g(y))$ is always undefined. So in a lazy functional language like Miranda, the undefined functions are not natural transformations. This is one of the reasons why categorical (and type-theoretic) treatments of computation cannot deal with non-terminating functions.

(3) The converse problem appears in a language like CSP, in which a program $p$ may engage in observable input and output before diverging. Thus $p; \perp \vec{\vec{p}} q$ behaves better than $\perp \overleftarrow{p} q; q$, which diverges immediately,

(4) In a strict functional or procedural language, there is a similar problem with the projection functions $0_{bc}$ and $1_{bc}$ for a cartesian product. If $q$ fails to to terminate, then $(p \times q); 1\vec{p}\vec{q}$ will attempt to execute $q$, and also fail; whereas $1\overleftarrow{p}\overleftarrow{q}; (p \times q)$ will succeed whenever $p$ does. So the projection functions are not natural transformations.

(5) In a non-deterministic programming language like Dijkstra's (or CSP), there is a similar problem with the operator $\Delta b$. If $h$ is a non-deterministic program, $\Delta \overleftarrow{h}; (h \times h)$ may deliver a pair $(x, y)$ of results in which $x \neq y$; whereas the results delivered by $(h; \Delta \overrightarrow{h})$ are always equal.

The solution to all these problems is to introduce an ordering relation $\sqsubseteq$ on the homsets of the category, and introduce a wider choice of abstraction functions for use in data refinement. That will be done in a later paper.

Acknowledgements

to Wim Hesselink, He Jifeng and Samson Abramski for assistance, encouragemen and advice of various kinds. Also to the Admiral R Inman Chair in Computer Science at the University of Texas at Austin for support of a more material kind during the studies which led to this paper.