Natural Transformations and Data Refinement

C.A.R. Hoare and He, Jifeng
Oxford University Computing Laboratory

Summary: Natural transformations play the role of retrieve functions in program development by data refinement [6]. A sufficient condition of validity of data refinement for a given programming language is that its constructors should be covariant endofunctors or (more generally) natural transformations between them.

Keywords : Data refinement, natural transformation, covariant functor

1. Introduction

Data refinement is one of the best known methods for the development of correct, efficient and even large programs. Much research in this area has produced various kinds of refinement rules [2,3,5,6,8]. It starts with a finite graph D, whose nodes are the names of types and whose arrows are the names of operations between them. These are first given an abstract interpretation A, which is used to construct a correct program. The types and operations are then given a concrete interpretation C, which may be more complex and more efficiently executed than A. The replacement of A by C is valid if there exists for each type name t a "retrieve" function ηt, mapping its concrete interpretation Ct to its abstract interpretation At. For each operation name p, the effect of applying its concrete interpretation Cp followed by the retrieve function must be the same as applying the retrieve function followed by the corresponding abstract operation Ap.

In category theory, the abstract and concrete interpretations are functors from the graph D to some concrete category M of sets and functions, and the retrieve function is nothing but a natural transformation between these functors. This is usually expressed by the commuting diagram

$$\begin{tikzcd}
As \arrow[r, "Ap"] & At \\
Cs \arrow[u, "\eta s"'] \arrow[r, "Cp"] & Ct \arrow[u, "\eta t"']
\end{tikzcd}$$

The validity of this replacement of A by C in a concrete program is critically dependent on the properties of the programming language in which the program is expressed. It turns out that the properties that are required of the language are familiar categorical concepts, namely that the sequential (or functional) composition of the language should be a categorical composition, and that its other constructors should be either covariant endofunctors or bifunctors or natural transformations between them.

2. Programming Language Semantics

The method of data refinement can be generalised to prove correctness of an implementation of an entire programming language. For this purpose, it is convenient to take an algebraic approach to its semantics.

The features of a programming language fall into two classes [10]: an inner language consisting of primitive types and operations between them; and an outer language of constructors by which structured types are constructed from primitive ones, and structured programs from basic operations. The syntax and semantics of the inner language and the outer language can be treated independently by different techniques; this will allow a similar useful decomposition in the proof of correctness of an implementation.

The syntax and type constraints of the inner language are presented as a graph D, as described in the previous section. Its semantics can be given as a functor A from D to a concrete category M. An implementation C can be defined as a second such functor; and a natural transformation n is found between C and A. It is our hope and objective that the correctness of the whole implementation of the whole language will follow automatically from naturality of n in the category of graphs. . But this hope depends on certain properties of the outer language, as described below.

The syntax and type constraints of the outer language are specified as a heterogeneous signature Σ, containing symbols for all the constructors of the language, and sorts for all its types. The set of sorts may themselves be defined by means of the type constructors of the language. An algebraic semantics for the language is given by a set of equations E governing the constructors in Σ. This defines a category (or variety) V. Its objects are small categories that are closed with respect to syntactically consistent application of the constructors of Σ, and satisfy the equations in E; and its arrows , say $h: B \to C$, are Σ-homomorphisms preserving all the constructors of Σ in the sense that

$$h(\sigma(x_1, .., x_n)) = \sigma(hx_1, .., hx_n)$$

for all $\sigma \in \Sigma$ and x_i in the carrier of B. Let U be the forgetful functor from V to the category of graphs, and let F be its free adjoint, θ the adjunction and ε the counit.

$$\begin{tikzcd}
D \arrow[r, "A"] & UM
\end{tikzcd}$$

$$\begin{tikzcd}
FD \arrow[r, "FA"] \arrow[dr, "\theta A"'] & F(UM) \arrow[d, "\varepsilon_M"] \\
 & M
\end{tikzcd}$$

Given a choice of graph D, the algebraic semantics of the language is just the corresponding free algebra FD in the variety V. Given next a choice of functor A, the corresponding denotational semantics is uniquely defined as θA; its denotational property is guaranteed by the fact that it is an arrow in V.

3. Correctness of Implementation

Now let C be a concrete implementation of the inner language. The induced implementation of the whole language is θC. In order to prove the correctness of the implementation, we need to find a natural transformation θn from θC to θA:

$$\theta n : \theta C \dashrightarrow \theta A$$

But far better to rely on some general theorem that guarantees the existence of θn, without going to the trouble of finding it in each case. Clearly a sufficient condition for this theorem is that the domain of the adjunction θ can be extended to all natural transformations n in a manner which preserves the commutativity of the diagram

$$\begin{tikzcd}
D \arrow[r, bend left, "A"{name=A}] \arrow[r, bend right, "C"'{name=C}] & UM \arrow[from=C, to=A, Rightarrow, "n"]
\end{tikzcd} \qquad \text{in GRAPH}$$

$$\begin{tikzcd}[column sep=large, row sep=large]
FD \arrow[r, bend left, "FA", ""'{name=U}] \arrow[r, bend right, "FC"', ""{name=L}] \arrow[dr, bend left=15, "\theta A", ""'{name=TA}] \arrow[dr, bend right=30, "\theta C"', ""{name=TC}] & F(UM) \arrow[d, bend right, "\varepsilon_M"', ""{name=E1}] \arrow[d, bend left, "\varepsilon_M", ""'{name=E2}] \\
& M
\arrow[from=L, to=U, Rightarrow, "Fn"]
\arrow[from=TC, to=TA, Rightarrow, "\theta n"]
\arrow[from=E1, to=E2, Rightarrow, "\varepsilon_M"]
\end{tikzcd}$$

in V

In this diagram, the counit ε_M is an identity natural transformation (effectively a functor), so θn can be simply defined as the horizontal composition of Fn and ε_M. In this case we say that the variety V respects natural transformations. By extension we apply the adjective "respectful" to the adjunction θ, the free functor F, the forgetful functor U, the constructors of the signature Σ and/or the equations E.

We have assumed throughout that our programming language contains some form of functional or sequential composition. This means that composition (here denoted by ;) will be a sort-indexed family of the operators of Σ (or be definable in terms of them). In order to establish validity of data refinement, our first requirement is that composition should satisfy the familiar categorical axioms; these should therefore be included in (or be derivable from) the equations E. This means that the objects of the category V are themselves categories, with additional structure representing the additional features of the language.

We can therefore investigate the categorical properties of the other constructors of Σ. A constructor is said to be an endofunctor if E implies the equations and type constraints which define a covariant functor, i.e., it distributes through composition. A constructor is defined to be a natural transformation between two such endofunctors if its naturality is derivable from E. Functors can be conventionally equated with their identity natural transformations. The achievement reported in this letter is the proof that all such natural transformations are respectful of natural transformation. This is a result of such reflexive elegance that we were surprised not to find it in standard texts on Category Theory.

4. Programming Languages

It is also an extremely useful result, because it ensures the validity of data refinement for all programs expressed in a programming language whose constructors are natural transformations. Many of the constructors of a typed programming language enjoy this property, for example

0. The constuctor of cartesian products (records in PASCAL) is a bifunctor.
1. The projections which access the components of a tuple are natural transformations, and so is the duplicating operation Δ which maps every y onto (y, y).
2. The constructor of discriminated unions (variant record in PASCAL) is a bifunctor.
3. The conditional or case statement of PASCAL can be defined in terms of natural transformations, namely the injections of the coproduct and the merging operator ∇, which maps both $(0, y)$ and $(1, y)$ to y.
4. The zero morphisms represent the undefined operator **abort** of Dijkstra's language [1]. They too are natural transformations.

However, there are certain constructors which are not natural.

0. Recursion cannot be modelled as a natural transformation.
1. The function-space constructor in a higher order language is a bifunctor, but it is contravariant in its first argument.
2. Non-determinism cannot be modelled as a natural transformation.
3. In a nondeterministic language, the duplicating constructor Δ is not a natural transformation.
4. In a strict language like PASCAL, the field selectors are not natural.
5. In a lazy functional programming language like Miranda, the non-terminating function $\perp$ is not natural.

The solution to some of these problems is to restrict the class of retrieve functions to total surjections or even bijections (natural isomorphisms). A more general solution is to introduce a partial ordering (like a Scott Domain [9]) into the theory. The results of this remain to be reported.

5. Proof outline

In this section, we shall use the letters c, d for objects, p, q, j, k, l for arrows, We shall use $^{<}p$ to denote the source of p, and $p^{>}$ the target. We shall equate an object with its identity arrow, even in the case of a graph.

The first programming language constructor which we shall consider is composition (functional or sequential); and to begin with, we assume that this to be the only constructor of the language L. So the program texts of L constitute the smallest set which contains the names from D; and whenever it contains k and l with $k^{>}={}^{<}l$ 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). Formally

$$j;(k;l)=(j;k);l$$
$$l;l^{>}={}^{<}l;l=l$$

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 FD over graph D within the category of all categories. This means that (as long as D 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 FD is the following:

Let A be any graph morphism from D to M (or more strictly UM, where U is the forgetful functor from the variety V to the category of graphs). Then there is an unique functor θA, whose domain is FD, and which agrees with A on D, i.e.,

$$\theta A(k;l)=\theta Ak;\theta Al \qquad \text{for } k.l\in FD$$
$$\theta Ap = Ap \qquad \text{for } p\in D$$

The fact that θA 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 seperated by semicolons. From the fact that θA distributes through semicolon, it follows that for any program text k in L, θAk is just the function in M which is obtained by interpreting each primitive name p of D within text k as the mathematical operation θAp, and taking semicolon to mean functional composition. In other words, θA (or indeed, any other functor) provides a semantics for the language which is denotational, in the sense that the meaning of the whole is given in terms of the meaning of its parts.

Now we can 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 $Cp;np^{>}=n^{<}p;Ap$ for all p in D then $\theta Ck;nk^{>}=n^{<}k;\theta Ak$ for all k in FD

Proof. by induction on the structure of k.

Base case: follows from the assumption because $\theta Cp=Cp$ and $\theta Ap=Ap$ for all p in D.

Induction step:

$$
\begin{array}{ll}
\theta C(p;q);n(p;q)^{>} & \\
= (\theta Cp);(\theta Cq);nq^{>} & \{\ \theta C \text{ is a functor, } (p;q)^{>}=q^{>}\} \\
= (\theta Cp);n^{<}q;(\theta Aq) & \{\text{induction hypothesis}\} \\
= n^{<}p;(\theta Ap);(\theta Aq) & \{\text{same again, } p^{>}={}^{<}q\} \\
= n^{<}(p;q);\theta A(p;q) & \{\ \theta A \text{ is a functor, } {}^{<}(p;q)={}^{<}p\} \qquad \square
\end{array}
$$

In the following we shall enrich our programming language with more and more constructors. So it seems advisable to generalise the previous result to deal with an arbitray set Σ of constructors, one of which is composition. We will adopt a consistent notation in the later proofs. Let θ be the adjunction associated with the forgetful functor U from a variety to the category of graphs, and let F be its left adjoint. Let $n:C\dot{\to}A$ be a natural transformation from functor C to functor A. We now need to define θn as the extension of n to objects of FD, in such a way that we can later prove it to satisfy $\theta n:\theta C\dot{\to}\theta A$. The definition of θn proceeds, in the same way as the definition of θC and θA, by induction on the constructor structure of the elements of FD.

$$
\begin{array}{ll}
\theta n(p) = np^{>} & \text{for all objects } p \text{ in } D \\
\theta n(fp) = f(\theta n(p^{>})) & \text{for } f \text{ in } \Sigma,\ p \text{ in } FD \text{ whenever } fp \text{ is an object.}
\end{array}
$$

The validity of this definition is not obvious. Firstly, an element of FD may have distinct representations fp and gq. This can only be because the equation

$$fp=gq$$

is provable from the equations of the Σ-variety. Consequently, this equation will still be true when every variable denoting an element p of D is replaced by $\theta n(p^{>})$. But that is exactly the same effect as applying θn to both fp and gq. So it makes no difference which of the two representations is used to define θn. Secondly, the right hand side of the defining equations may violate type consistency. We will solve this problem by confining attention to functors f, which are always defined on every elements of FD. Note that natural transformations do not introduce any new objects into a category; that can be done only by the functors that they relate. So it is not necessary to extend θn in these cases.

Theorem 2.

Endofunctors respect natural transformation.

Proof: let Σ contain only composition and endofunctors. We use induction on

the structure of elements of FD.
Base case: assume $p \in D$, so also $p^{>} \in D$

$$\begin{aligned}
& \theta C p ; \theta n p^{>} \\
&= C p ; n p^{>} && \{\theta C \text{ extends } C, \theta n \text{ extends } n\} \\
&= n^{<} p ; A p && \{n : C \dot{\rightarrow} A, \text{ by assumption}\} \\
&= \theta n^{<} p ; \theta A p && \{\theta A \text{ extends } A, \theta n \text{ extends } n\}
\end{aligned}$$

Induction step: let $f \in \Sigma$, and consider $fp \in FD$

$$\begin{aligned}
& \theta C(fp) ; \theta n((fp)^{>}) \\
&= f(\theta C p) ; \theta n(f p^{>}) && \{\theta C \text{ is } \Sigma\text{-homomorphism, and } f \text{ is a functor}\} \\
&= f(\theta C p) ; f(\theta n p^{>}) && \{\text{ definition of } \theta n \text{ for constructor } f\} \\
&= f(\theta C p ; \theta n p^{>}) && \{f \text{ is a functor}\} \\
&= f(\theta n^{<} p ; \theta A p) && \{\text{induction hypothesis}\} \\
&= \theta n(^{<}(fp)) ; \theta A(fp) && \{\text{by a mirror argument}\} \qquad \square
\end{aligned}$$

The next theorem extends the previors result to Σ containing both endofunctors and natural transformations between them.

Theorem 3.
Natural transformations respect natural transformations.
Proof: let f and g be endofunctors in Σ and let m in Σ be a natural transformation between them. The main part of the proof is the same as that of Theorem 2. We need consider additionally those elements of FD which can be expressed in the form md, where d is an object of FD.

$$\begin{aligned}
& \theta C(md) ; \theta n(md)^{>} \\
&= \theta C(md) ; \theta n(gd) && \{md : fd \rightarrow gd\} \\
&= m(\theta C d) ; g(\theta n d) && \{\theta C \text{ is a } \Sigma\text{-homomorphism, and } m \in \Sigma, \text{ def of } \theta n\} \\
&= m(^{<}(\theta n d)) ; g(\theta n d) && \{\text{by induction hypothesis: } \theta n d : \theta C d \rightarrow \theta A d\} \\
&= f(\theta n d) ; m((\theta n d)^{>}) && \{m : f \dot{\rightarrow} g, \text{ by assumption}\} \\
&= \theta n^{<}(md) ; \theta A(md) && \{\text{by a mirror argument}\} \qquad \square
\end{aligned}$$

A very similar proof applies to covariant endofunctors of higher arity. The crucial property of such functors is that they admit distribution by θn to all their arguments. So do arbitrary combinations of such endofunctors. So do certain other functors such as the identity functor, or the selector functors of higher arity which select a single one of their arguments, e.g.

$$\begin{aligned}
&\text{if } 1^{st} pq = p, \quad 2^{nd} pq = q, \text{ for all } p, q \\
&\text{then } \theta n(1^{st} pq) = \theta n p = 1^{st}(\theta n p)(\theta n q) \\
&\text{and } \theta n(2^{nd} pq) = \theta n q = 2^{nd}(\theta n p)(\theta n q)
\end{aligned}$$

DATA REFINEMENT IN A CATEGORICAL SETTING.

C.A.R. HOARE, June 1987.

Data refinement is one of the most effective formal methods for the step-wise development of large programs and systems. The system design is expressed as a program text, which is first interpreted as operating on data of abstract type. The simple mathematical properties of abstract data are helpful in deriving the design from its specification. At the next step, the abstract data are represented compactly by bit-patterns (say) in the store of a computer, and the required operations upon them are implemented by efficient subroutines. The same program text developed in the previous step is then given this new concrete interpretation, so that it can be executed directly by a computer. In the case of a large system, the transition between design and code is split into many steps, each of which provides the starting point for the following step.

The correctness of the more concrete interpretation is established by a collection of abstraction functions, which map each concrete type to the corresponding abstract type. Each abstraction function must be proved to commute with each primitive operation of the appropriate data type, in the following sense:

> To apply the abstraction function after any concrete operation gives the same (or better) result as applying it before the corresponding abstract operation.

This fact is proved only for the primitive operations invoked by the program; as a consequence, it is valid for any program written using those primitives, provided that the programming language has been designed with sufficient care. This paper investigates the conditions under which data refinement is a valid method for program development.

Summary (only for category theorists)

The relevance of category theory to data refinement is suggested by the uniform view which both of them take towards data types and operations on values of each type. The advantage of the categorical setting lies in its purely algebraic proofs, which do not need to mention the individual data

values of each type.

The programs in a strictly typed programming language form a category L, in which composition is just the familiar sequential composition of programming, denoted by semicolon. An abstract interpretation of the language L is given by a functor G , which maps each program of L into some mathematical category M . The functorial property of G ensures that it respects the original type structure as well as the syntactic structure of the program. A concrete interpretation is given similarly by a different functor F , which maps L into some (usually) different part of the same category M. Now the abstraction function of data refinement is nothing other than a natural transformation between these two functors.

Among the elements of a programming language L we can single out a subset L_0 containing just the primitive data types and the built-in operations upon them. The combinators of the language (for example, sequential composition) are called generators, because every program in L can be generated by a finite number of applications of the generators to the primitive elements of L_0 . A generator "g" in L is a function from L to L , written in quotes to emphasise its syntactic nature. It is assumed to have a mathematical meaning g , which is a function from M to M . Any interpretation of the language L must respect this meaning, in the usual sense of denotational semantics. So we require that all functors F from L to M must commute with every "g" in the sense that

$$F(\text{"g"}p) = g(Fp) \quad \text{for all } p \text{ in } L$$

A beneficial consequence of this requirement is that the whole meaning of a functor can be defined by just giving its value when applied to elements of L_0 . Its value on any generated element of L can then be computed by primitive recursion on the structure of the generation tree.

Similarly, we want to prove the commuting property of a natural transformation only for types and operations in L_0 , and on the basis of this simple proof, we want to be sure that the commuting property holds for all generated programs in L . This is what is meant by the statement that natural transformations are valid for data refinement. The purpose of this research is to explore the design constraints on the language L which will maintain this validity.

Let f be a partial function from the semantic category M to itself.

Suppose we wish to insert f as a new feature in our programming language. It is mathematically trivial to choose a new notation "f" to denote the function, and to insert it among the generators of L , subject to the same type constraints in L as f is subject to in M . This will enlarge the class of texts in the language to include those which mention "f" in a syntactically valid and type-consistent way. If every natural transformation valid on the original smaller language is still valid on the extended language, we say that the extension preserves the validity of natural transformation. If the introduction of "f" can generate new identities (objects) in L , the definition of the natural transformation n must be extended to these new identities as well. This is done by the usual commuting equation

$$n(\text{"f"}b) = f(nb) \quad \text{for all identities } b \text{ of } L .$$

The main result of this paper is to show that any functor from M to itself will preserve validity of natural transformations, and that any natural transformation between such functors will do so too. This is a theorem of such elegance that it must be a special case of some more general theorem known to categorists but not to me.

The importance of the result is that many of the features that we want and find in a programming language are either functors or natural transformations. However, to deal with languages which contain non-terminating or non-deterministic programs, we will need to introduce a slight generalisation of the natural transformation, known as a simulation. It is explained in the remainder of this section.

In program development, it is not necessary to insist on absolute identity of the effects of the concrete and abstract programs. It is certainly enough to require that the concrete program is better than the abstract one in all relevant respects, and in all contexts of use. We therefore introduce a partial ordering $\subseteq$ (pronounced "upward") into our categories, to denote that the left operand is an improvement on the right operand (which must have the same domain and codomain). Here are two of the ways in which a program p may be uniformly as good or better than q:

(1) p terminates and gives the same result as q (or better) in all cases that q terminates (but perhaps p also terminates in cases that q may fail).

(2) every result that p can give is the same as (or better than) some

result that q can give (but q can give a wider range of different results). Thus p is more predictable, more controllable, and more deterministic than q.

In the mathematical theory, $\sqsubseteq$ is an arbitrary partial order, and may be interpreted as any kind of improvement. To ensure that the improvement is maintained in all contexts, we postulate that all operators, combinators, and functors are monotonic.

Now the commuting equation defining naturality can be replaced by an inequation, expressing the superiority of the concrete functor. This can be done in two different ways, leading to two definitions.

(1) An upward simulation u is defined as a transformation from F to G such that

$ub: Fb \to Gb$,	for all identities b in L
$Fp ; ub' \sqsubseteq ub ; Gp$,	for all $p: b \to b'$ in L.

(2) A downward simulation d is defined as a transformation from G to F such that

$db : Gb \to Fb$,	for all identities b in L
$db ; Fp \sqsubseteq Gp ; db'$,	for all $p : b \to b'$ in L.

Clearly, the familiar natural transformation is both upward and downward from F to G. Another way of combining the two definitions is in the definition of a total simulation. This is a pair (d,u), where

1. u is an upward simulation
2. d is a downward simulation
3. $ds ; us = Gs$ and $Fs \sqsubseteq us ; ds$

A total simulation establishes a pre-order in a category, in the same way as a natural isomorphism establishes an equivalence. The preorder is the one used by Scott to find a solution for reflexive domain equations.

In a simple category L, all three kinds of simulation are valid, in the sense that the simulation property needs proof only on the generating graph L_0. But a simple category is a rather weak programming language, in which only straight line programs can be written. This paper investigates a series of generators which enrich the category L, including least upper bounds, zero morphisms, coproducts, products or smash products, and higher order function spaces (cartesian closure). The same enrichments are made to the semantic category M , and all functors are assumed to respect the additional structure. Each enrichment is treated separately, so that the proofs apply to the widest possible variety of languages. For some of the enriched languages, both kinds of simulation are valid, and for others, only one is valid. Total simulation is the only valid method for all cases.

The most characteristic feature of a general-purpose programming language is recursion, in some languages confined to a special iterative form. The meaning of recursion can be given by allowing generations to be applied a countable number of times, thus generating infinite expressions, or trees. A recursively defined program unit

$$X = FX$$

is then identified with its infinite unfolding. This gives a sort of operational semantics for recursion.

In category theoretic terms, this is the "cofinal algebra" semantics. Equality (or ordering) between trees can be defined in terms of the ordering of all finite "prunings", and so can be proved by induction (and must be, because equality is no longer decidable). Thus the inductive proofs establishing validity of data refinement will hold (I think) for recursive programs too. Perhaps further research is called for here.

An interesting by-product of this research is an understanding how a category provides an algebraic semantics for a range of programming languages, even those which include non-termination, non-determinacy, higher order procedures, and a limited form of concurrency.

Introduction to category theory (for computing scientists)

We define a graph to be a set G with two monadic operators (total functions from G to G)

domain, denoted by prefix $^<$

codomain, denoted by postfix $^>$

These operators bind even tighter than function application. They are assumed to satisfy the following axioms

$$(p^{>})^{>} = p^{>} = {}^{<}(p^{>})$$

$${}^{<}({}^{<}p) = {}^{<}p = ({}^{<}p)^{>}$$

Consequently, both operators have the same range (image), whose elements are known as identities. They are elsewhere called nodes or objects, and they represent the data types of a programming language. In a procedural language, they also represent the structure of the machine state or stack during execution. They will be denoted by early letters in the alphabet – b, c, d. We also use the abbreviation

$$p : b \rightarrow c \quad \text{means} \quad {}^{<}p = b \text{ and } p^{>} = c$$

It is easy to prove that

p is an identity

iff $p^{>} = p$ (or equivalently, ${}^{<}p = p$) .

A graph morphism is defined as a function from one graph to another, provided that it preserves the graph structure; in other words, it commutes with the domain and codomain operators

$$fp^{>} = (fp)^{>} \quad \text{and} \quad f{}^{<}p = {}^{<}(fp)$$

Clearly, a graph morphism maps identities to identities.

A category C is a graph together with a partial dyadic function known as composition, and denoted here by infix semicolon, which binds less tightly than function application. The following axioms must also be satisfied

$$p;q \text{ is defined if and only if } p^{>} = {}^{<}q$$

$$(p;q)^{>} = q^{>} \quad \text{and} \quad {}^{<}(p;q) = {}^{<}p$$

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

$$p;p^{>} = p = {}^{<}p;p$$

If identities are taken to be null commands (e.g., "skip"), then sequential composition in a normal programming language clearly satisfies these axioms. It is defined only if the type of the result of the first operand is the same as the type expected initially by the second operand.

A partial order $\sqsubseteq$ (pronounced "upward") is defined to be a relation which is reflexive, transitive, and antisymmetric. A partial order on a category holds only between elements of the same type; and composition is monotonic

$$p \sqsubseteq q \Rightarrow p^{>} = q^{>} \text{ and } {}^{<}p = {}^{<}q$$

$$p \sqsubseteq q \Rightarrow p;r \sqsubseteq q;r \text{ and } r;p \sqsubseteq r;q$$

Clearly, equality itself satisfies these axioms; and so does the converse of $\sqsubseteq$, which will be denoted $\sqsupseteq$ and pronounced "downward". We will henceforward be concerned with categories ordered by $\sqsubseteq$, $\sqsupseteq$, and $=$. Conventional category theory is the special case where these three orderings are the same.

A retraction is defined as a pair (d,u) of elements of a category, where

$$d;u = {}^{<}d = u^{>}$$

$$u;d \sqsupseteq {}^{<}u = d^{>}$$

The next theorem shows that each element of a retraction uniquely determines the other

Theorem 0. Let (d,u) and (e,v) be retractions. Then

$$d = e \text{ iff } u = v$$

Proof: assume $d = e$

because ${}^{<}u \sqsubseteq u;d$, $d^{>} = {}^{<}u$, and composition is monotonic

$${}^{<}u;v \sqsubseteq (u;d);v = (u;e);v$$

by cancellation of identity

$v \subseteq (u;e);v$

composition is associative

$v \subseteq u; (e;v)$

$(e;v)$ is an identity and can be cancelled

$v \subseteq u$

The proof that $u \subseteq v$ is similar.

The proof of the reverse implication is similarly similar

end of proof.

The next theorem shows that compatible retractions can be composed

<u>Theorem 1</u>. If (d,u) and (e,v) are retractions, and $e^{>} = {}^{<}d$, then $(e;d, u;v)$ is a retraction.

Proof: $(e;d) ; (u;v)$

composition is associative

$= e; (d;u) ; v$

(d,u) is a retraction

$= e; {}^{<}d; v$

cancellation of identity, and $e^{>} = {}^{<}d$

$= e; v$

(e,v) is a retraction

$= {}^{<}e$

The other half of the proof is similar, using inequations and monotonicity of composition.

end of proof.

A total monotonic function F from category L to category M is said to be a functor (abbreviated $F : L \to M$) if it is a graph morphism that distributes through composition

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

A functor from M to itself is known as an endofunctor. The next theorem shows that functors can be composed.

Theorem 2. Let $H : M \to N$ Then the composition HoF is also a functor from L to N.

Proof: $((HoF)p)^{>}$

by definition of composition o of functions

$$= (H(Fp))^{>}$$

H is a functor

$$= H(Fp)^{>}$$

F is a functor

$$= H(Fp^{>})$$

definition of o

$$= (HoF)p^{>}$$

The proof for $^{<}$ is similar. Now consider semicolon

$$(HoF)(p;q)$$

definition of o

$$= H(F(p \; ; q))$$

F is a functor

$= H(Fp; Fq)$

H is a functor

$= H(Fp) ; H(Fq)$

definition of o (twice)

$= (HoF)p ; (HoF)q$

end of proof.

Let F and G be two functors from L to M , and let t be a function from the identities of L to the elements of M . Then t is said to be a transformation from F to G if its domain agrees with F and its codomain with G

$^{<}(tb) = Fb$ and $(tb)^{>} = Gb$ for all identities b in L

If furthermore

$Fp ; tp^{>} \underline{c} \; t^{<}p ; Gp$ for all p in L ,

then t is called an $\underline{c}$-simulation (abbreviated $t : F \underline{c} G$). A $\underline{d}$-simulation $d : G \underline{d} F$ is defined similarly. A natural transformation n is defined as a simulation that is both upward and downward from F to G .

A total simulation from F to G is a pair (d,u) , where

(1) (db ; ub) is a retraction in M , for all identities b of L

(2) $u : F \underline{c} G$

(3) $d : G \underline{d} F$

Either of the conditions (2) and (3) could be omitted, in the light of the important theorem

<u>Theorem 3.</u> (1) => ((2) $\equiv$ (3))

Proof: first assume (1) and (2)

$d^{<}p; Fp$

insertion of redundant identity, since $^{<}(up^{>}) = (Fp)^{>}$

$d<p; Fp ; {}^{<}(up^{>})$

(d; u) is a retraction, composition is monotonic and

$\subseteq (d^{<}p; Fp); (up^{>}; dp^{>})$

composition is associative

$= d^{<}p; (Fp; up^{>}); dp^{>}$

by assumption (2) and composition is monotonic

$\subseteq d^{<}p; (u^{<}p; Gp); dp^{>}$

composition is associative

$= (d^{<}p; u^{<}p); (Gp; dp^{>})$

assumption (1)

$= Gp; dp^{>}$

The other half of the proof of (2) from (3) and (1) is similar.

end of proof.

This theorem greatly reduces the labour of using total simulations, because it allows proof of the commuting property of only one of the simulations say u . Then if u is (for example) a total surjective function, it is known to have a unique partner d such that (d,u) is a retraction. So if u has been proved to be an upward simulation, and is a total surjective function, it is in effect also a total simulation.

A simulation is an appropriate method of connecting two functors, both mapping a category L to a category M . We now consider two functors which map in opposite directions

$V : L \to M$

$U : M \to L$

We define a method of connecting these two functors which will be known as a rightward junction from V to U. It is a function Θ of three arguments; the first is an identity in L, the second is an identity in M, and the third is an element in L. The result of ~~f~~ Θ is an element of M. The defining properties of a junction are

0. If $q : b \to Uc$ in L

   then $\Theta bcq : Vb \to c$ in M

1. $\Theta^{<}ps^{>}(p ; q ; Us) = Vp ; \Theta p^{><}sq ; s$

If p is an identity, property 1 simplifies to

1a. $\Theta^{<}qs^{>}(q ; Us) = \Theta^{<}q^{<}sq ; s$

and if s is an identity

1b. $\Theta^{<}pq^{>}(p ; q) = Vp ; \Theta p^{>}q^{>}q$

for all p, q in L, and s in M

A leftward junction $\Theta\sim$ from U to V is defined similarly:

0. If $r : Vb \to c$ in M

   then $\Theta\tilde{}bcr : b \to Uc$ in L

1. $\Theta\tilde{}^{<}ps^{>}(Vp ; r ; s) = p ; \Theta\tilde{}p^{><}sr ; Us$ for all p in L and r, s in M

If $\Theta\tilde{}$ is the inverse of Θ, ie,

$\Theta\tilde{}bc(\Theta bcp) = p$ for all p in L

and $\Theta bc(\Theta\tilde{}bcr) = r$ for all r in M

then the bijection $(\Theta,\Theta\tilde{})$ is known as an adjunction in category theory. Further V is called the left adjoint and U the right adjoint of the adjunction.

Validity of simulation

Composition is the first and most important of the operations of category theory, and it is present as a generator in almost all programming languages. Our first task is therefore to prove that it preserves the validity of each of the three kinds of simulation. That means that a simulation that has been proved to commute for all elements of the graph L_0 will still commute on additional elements of L, ie., the sequences obtained by repeated composition . As might be expected, the proof uses an induction hypothesis that each operand of the composition satisfies the commuting property.

Theorem 4. Introduction of composition maintains validity of each kind of simulation.

Proof (for upward simulation).
Every new element is of the form $p;q$, where $p^{>} = {}^{<}q$

$$F(p;q) \,;\, u(p;q)^{>}$$

F is a functor, and property of composition

$$= Fp \,;\, Fq \,;\, uq^{>}$$

induction on q , and composition is monotonic

$$\subseteq Fp \,;\, u^{<}q \,;\, Gq$$

composition is defined

$$= Fp \,;\, up^{>} \,;\, Gq$$

induction on p , and composition is monotonic

$\underline{c}$ $u^{<}p$; Gp ; Gq

property of composition, and G is a functor

= $u^{<}(p;q)$; G(p;q)

end of proof.

The domain and codomain of (p;q) are the same as those of p and q respectively. So composition cannot introduce any new identities into the category, and the definition of a simulation does not need to be extended.

Composition of simulations

A most valuable aspect of data refinement is that it may be applied repeatedly in many steps throughout the design of a complex system. At each step, a simulation is proved to connect the result of the previous step to the input of the next one. Assuming that all simulations are of the same kind, the correctness of the stepwise process is established by composing the whole sequence of successive simulations into a single simulation, which connects the design of the first step to the code of the last. This composition is defined in the obvious way, and is obviously associative

(u;v)b = (ub;vb) and (e;d)b = eb;db

where u: F $\underline{c}$ G , v: G $\underline{c}$ H

e: H $\underline{d}$ G , d: G $\underline{d}$ F

<u>Theorem 5</u>. u;v is a simulation of the same kind as u and v

Proof: (for upward simulations u, v)

Fp ; $(u;v)p^{>}$

definition of composition of simulations

= Fp; $up^{>}$; $vp^{>}$

u is upward from F to G and compositon is monotonic

$\subseteq u^{<}p; Gp; vp^{>}$

v is upward from G to H

$\subseteq u^{<}p ; v^{<}p ; Hp$

definition of composition of simulations

$= (u; v)^{<}p; Hp$

end of proof.

The composition of total simulations is defined

$((d,u); (e,v))b = ((e;d)b, (u;v)b)$

Theorem 6. The composition of total simulations is a total simulation.

Proof. By theorem 1, the composition is a retraction. By theorem 5, the component (u;v) is an upward simulation. By theorem 0, (e;d) is uniquely determined, and by theorem 2 it is a downward simulation.

A simple generator

We turn now to our main task of considering what functions on M can be included into the programming language L , while preserving the validity of data refinement. Consider a function t from the identities of M to the elements of M , which has the following two properties:

0. $tb : b \to b$

1. $p ; tp^{>} = t^{<}p ; p$

In other words, t is a natural transformation from the identity functor to itself.

An uninteresting example of such a transformation is the identity function (tb = b for all b). A more interesting example is the function that maps each data type to the **abort** command (on data of the same type) . Among the many defects of **abort** is the possibility that in all initial conditions it will fail to terminate. Property 1. is satisfied in Dijkstra's programming language, because

p ; **abort** = **abort** ; p

In words, a program which starts by failing to terminate is indistinguishable from one which ends by failing to terminate.

In a category with zero morphisms, tb could be defined as $0bb$, the zero morphism between b and b. This would satisfy the additional axiom

$tb ; p = tb ; q$ for all $p, q : b \to b$

This law is also true for **abort** in programming languages, and so is the law which states that **abort** is the worst of all programs

$p \subseteq t^{<}p ; p$ for all p.

However, our main concern is data refinement, which does not rely on these two additional laws.

The function t can be introduced into the programming language with the notation "t", which is designed to have the same typing property O as t. Because of this, it cannot introduce any new identities into the language (by Property O, $(\text{"t"}b)^{>} = \text{"t"}b \Rightarrow \text{"t"}b = b$).

<u>Theorem</u>. t preserves the validity of all kinds of simulation.

Proof. $F(\text{"t"}b) ; u(\text{"t"}b)^{>}$

functors distribute through generators, and property 0 of "t"

$= t(Fb) ; ub$

u is a transformation from F to G

$= t^{<}(ub) ; ub$

property 1 of t

$= ub ; t(ub)^{>}$

u is a transformation from F to G

$= ub ; t(Gb)$

property 0 of "t", and G distributes through generators

$$= u^{<}(\text{"t"}b) ; G(\text{"t"}b)$$

end of proof.

A language like CSP contains commands for input and output, which have results observable before the program terminates (or fails to do so). Consequently, the aborting command (CHAOS) does not satisfy property 1. However it has the weaker property that non-termination after performing the inputs and outputs of p cannot be worse than immediate non-termination. So for CSP, property 1 must be replaced by

$$p ; tp^{>} \subseteq t^{<}p ; p$$

This states that t is an upward simulation from the identity functor to itself.

This weakening invalidates upward simulation. But downward simulation remains valid. The proof is the same as the one given above, except that the equation justified by property 1 is replaced by the downward inequation. As a result, total simulation remains valid. The reason is that the downward component is valid, and the other component is still upward because of the retraction property.

In a functional programming language composition denotes functional composition. If the language has a semantics based on lazy evaluation, a function (such as a constant function) can be evaluated without evaluating its argument. As a result, it will terminate even when applied to a non-terminating argument. However, the wholly undefined function always fails. On the principle that failure is worse than any kind of success, property 1. has to be replaced by

$$\textbf{abort} ; p \subseteq p ; \textbf{abort}$$

In such a language, the corresponding t is a downward simulation, and it is downward simulation that is no longer valid. In a language which combines the possibility of non-termination, a lazy evaluation strategy, and synchronised com-munication, neither of the above inequations will hold; and data refinement proofs will be more difficult.

Functional generators

The t introduced in the previous section was defined only on the identities of L_o. We now consider a monotonic function f defined on all elements, subject to the distributive properties

0. $fp : {}^{<}p \to p^{>}$

1. $f(p;q;r) = p;fq;r$

In other words, f is a junction from the identity endofunctor to itself. Introduction of such an "f" preserves the validity of all kinds of simulation. As before, the proof considers only elements of the form "f"p , but now it is necessary to use the induction hypothesis that d is a simulation of the same kind on p .

Proof(for downward simulation).

$G(\text{"f"}p) ; d(\text{"f"}p)^{>}$

functors distribute through generators, and property 0 of "f"

$= f(Gp) ; dp^{>}$

property 1 of f (the missing component is an identity)

$= f(Gp ; dp^{>})$

induction on p , and f is monotonic

$\underline{d}\ f(d^{<}p ; Fp)$

property 1 of f

$= d^{<}p ; f(Fp)$

property 0 of "f" , and functors distribute through generators

$= d^{<}(\text{"f"}p) ; F(\text{"f"}p)$

end of proof (for downward simulation).

If f is a function that somehow worsens its argument, it may be better to postpone the application of f as long as possible. Thus property 1 should be weakened to the chained inequations

$$p ; fq \subseteq f(p;q) \subseteq fp ; q$$

This weakening invalidates upward simulation but not downward or total simulation. The proof is the same as that given above, except that the lines justified by property 1 are replaced by inequations.

Similar reasoning applies to a dyadic function g , defined on pairs of elements with the same domain and the same codomain. An example of such a function is the non-deterministic or of a language such as CSP. This allows an implementation to make an arbitrary selection between the two operands. The distribution law is usually written in infix form

$$p;(q \text{ or } r);s = (p;q;s) \text{ or } (p;r;s)$$

This law states that it makes no difference whether the selection is made before execution of the first operand of a composition (e.g., at compile time), or whether it is made (at run time) after execution of the first operand.

Functorial generators

We now consider functions which obey a different set of distribution laws, namely the same laws which define a functor

0. $fp : f^{<}p \to fp^{>}$

1. $f(p;q) = fp ; fq$

The interesting feature of such generators is that when applied to identities they generate new identities. So we need to decide how to extend the definition of simulations, when applied to these generated arguments. This is done in the usual way by defining them to commute with the generator "f" in L

$$u(\text{"f"}b) = f(ub) \quad \text{and} \quad d(\text{"f"}b) = f(db) \quad \text{for all identities } b \text{ in } L$$

Theorem . For a total simulation, this preserves the retraction property

Proof. d("f"b) ; u("f"b)

by the definition given above

= f(db) ; f(ub)

f is a functor

= f(db ; ub)

by induction - (d,u) is a total simulation

= f(Gb)

G is a functor, and distributes through generators

= G("f"b)

The other half of the proof is similar, relying on monotonicity of f .

end of proof.

Theorem . A functorial generator preserves the validity of all kinds of simulation.

Proof. (for upward simulation)

F("f"p) ; $u("f"p)^{>}$

by distribution through generators, and property 0 of "f"

= f(Fp) ; $f(up^{>})$

f is a functor

= $f(Fp ; up^{>})$

f is monotonic, and induction hypothesis

$\underline{c}$ f($u^{<}p$; Gp)

by a mirror argument

= $u^{<}$("f"p) ; G("f"p)

The proof for a downward simulation is similar.

end of proof.

Similar arguments apply to a functor g with two parameters (known as a bifunctor), which is defined to satisfy the distribution laws

0. $^{<}(gpq) = g\,{}^{<}p\,{}^{<}q$ and $(gpq)^{>} = g\,p^{>}\,q^{>}$

1. g (p;q) (r;s) = gpr ; gqs

A simple example of a bifunctor is one that selects one of its operands

Gpq = p for all q

Proof:

$^{<}(Gpq) = {}^{<}p = G{}^{<}p{}^{<}q$ and similar for $^{>}$

G(p;q)(r;s) = p;q = Gpr; Gqs

end of proof.

A bifunctor may be converted to a single functor in any one of three ways

(1) fix its first argument to an identity

(2) fix its second argument to an identity

(3) identify its two arguments with each other

Proof: (1) let fq = gbq . Then

$^{<}(fq) = {}^{<}(gbq) = g{}^{<}b{}^{<}q = gb{}^{<}q = f{}^{<}q$ etc.

f(p;q) = gb(p;q) = g(b;b)(p;q) = gbp ; gbq = fp ; fq .

(3) Let $fp = gpp$. Then

$^{<}(fp) = {}^{<}(gpp) = g^{<}p^{<}p = f^{<}p$ etc.

$f(p;q) = g(p;q)(p;q) = gpp \,;\, gqq = fp \,;\, fq$

end of proof.

In fact, a functor in any number of variables taking values in a variety of categories, can be defined by composing any number of functors applied to those variables and to identities.

Simulation generators

The arguments in the section on zero morphisms generalise to simulations between any pair of functorial generators. For example, let t be a generator which is an upward simulation from functorial generator f to g .

Theorem . t preserves the validity of downward simulation.

Proof. $d^{<}(\text{"t"}b) \,;\, F(\text{"t"}b)$

"t" is a transformation from f to g

$= d(\text{"f"}b) \,;\, F(\text{"t"}b)$

distribution through generators

$= f(db) \,;\, t(Fb)$

d is a transformation from G to F

$= f(db) \,;\, t(db)^{>}$

t is upward from f to g

$\subseteq t^{<}(db) \,;\, g(db)$

by a mirror argument

$= G(\text{"t"}b) ; d(\text{"t"}b)^{>}$

end of proof

Corollary. A natural transformation, being a simulation in both directions, preserves validity of all types of simulation.

A similar argument applies to a simulation t between bifunctors f and g, which have the properties

0. $tbc : fbc \rightarrow gbc$

1. $fpq ; tp^{>}q^{>} \subseteq t^{<}p^{<}q ; gpq$

The definition of simulation is extended as usual to newly generated elements by distribution

$u(\text{"t"}bc) = t(ub)(uc)$

and all proofs go forward as before (I hope).

Discriminated Union

A familiar and useful example of a bifunctor is the one that forms the discriminated union $(b + c)$ of two data types b and c . This is sometimes known as the direct sum (in set theory), coproduct (in category theory), and appears as a variant record in PASCAL. A data value of type $(b + c)$ is a pair (tag, x), where

either (0) tag = 0 and x is of type b

or (1) tag = 1 and x is of type c

If $p: b \rightarrow b'$ and $q: c \rightarrow c'$, then $(p + q)$ represents a case statement which firsts tests the tag; if the tag is zero it executes p , or if the tag is 1 it executes q . The result of either execution is then tagged with the same value as initially. This gives a result in the right type, namely $(b' + c')$. But the tags are just representation details; they should be ignored in the mathematical theory.

The discriminated union provides a convenient method of modelling the familiar conditional construction of a programming language. For example, the test "even" , which tests whether a number is odd or even, can be regarded as a function from the natural number type IN to the disjoint union IN + IN . When applied to an even number, 2n , its result (0, 2n) is the same number tagged as the first alternative of the discriminated union; whereas an odd number is mapped into (1, 2n+1) , the same number tagged as in the second alternative. To halve a number if it is even, or add one if it is odd, can be achieved by the composition

even; (halve + succ)

But it still remains to map the result of this conditional from the discriminated union (IN + IN) back to the single natural number type IN . For this we need for each type b , a "merge" operator symbolised by Vb , which maps a disjoint union (b + b) onto the type b , simply by forgetting the tag which determines from which of the two (identical) types its argument has originated. Thus to achieve the effect

if even(x) **then** x := x/2 **else** x:= x+1 **fi**

the conditional described above should be completed as follows

even; (halve + succ) ; VIN

If p maps b to b' , p may be applied <u>after</u> the merging operation Vb , or it may be applied to both alternatives <u>before</u> the merging operation Vb' ; the final result of each of these applications will be the same. Thus merging satisfies the algebraic law

$$V^{<}p \; ; \; p \; = \; (p + p) \; ; \; Vp^{>}$$

<u>Theorem .</u> The merging operator preserves all kinds of simulation.

Proof: The algebraic law states that V is a natural transformation between the identity functor and the functor that maps p onto (p + p)
end of proof.

In a programming language, there are two extreme conditions for each pair of types b and c

tbc (meaning true) which always selects the first alternative (of type b) fbc (meaning false) which always selects the second alternative (type c)

Thus if (p + q) is executed after $t^{<}p^{<}q$, the first alternative p is invariably selected; so the effect is the same as if p had been applied beforehand

$$t^{<}p^{<}q \; ; (p+q) \;=\; p \; ; tp^{>}q^{>}$$

Similarly

$$f^{<}p^{<}q \; ; (p+q) \;=\; q \; ; fp^{>}q^{>}$$

These preserve validity of all kinds of simulation, because they are natural transformations from the bifunctor which selects one of its operands to the discriminated union bifunctor.

Here are additional laws which connect true, false and V

$$tbb \; ; Vb \;=\; b \;=\; fbb \; ; Vb$$

They are not necessary to the validity of data refinement.

Cartesian product

Another familiar and useful example of a bifunctor is the one that forms the cartesian product (b x c) of two data types b and c . This effect is achieved in PASCAL by a record declaration. A data value of type (b x c) is an ordered pair (x,y) where x is of type b and y is of type c . If p:b->b' and q:c->c' , then (p x q) is a command which executes p on the first component of the pair and q on the second component. The result is just the pair of results produced and so has the type (b' x c') . Since the components of a pair are disjoint, p and q can be executed serially in either order, or even concurrently. But that is an implementation detail, and can be ignored in the theory.

A frequently required operation on pairs is the selection of the first or second component. In PASCAL this is done by field names, and in LISP by car and cdr . We choose to make the types of the components explicit, and so introduce a pair of operators for each pair of data types b and c

$$\pi bc : b \times c \to b$$

$$\pi' bc : b \times c \to c$$

with the intention that

$$\pi(x,y) = x$$

and $\pi'(x,y) = y$

In category theory this intention must be expressed without mentioning individual values x and y . The required laws are mirror images to the laws for true and false described in the previous section

$$(p \times q) ; \pi p^{>} q^{>} = \pi^{<} p^{<} q ; p$$

$$(p \times q) ; \pi' p^{>} q^{>} = \pi'^{<} p^{<} q ; q$$

The left hand side of each equation describes the application of p to the first component, and the application of q to the second component of a pair; this is followed by discard of one of these results. The right hand side describes the more efficient program which discards the unwanted component first, and the performs only the appropriate operation. It seems reasonable to postulate that this optimisation does not change the meaning of the program.

But in many languages the equation does not hold. Suppose that the calculation on the discarded alternative fails to terminate. Then the execution of the left hand side may also fail to terminate. The right hand side does not involve an operation on the discarded alternative, and will therefore terminate in cases that the left hand side will not. This means that the right hand side in general can only be better than the left hand side, and so the optimisation mentioned in the previous paragraph is still valid. This is expressed mathematically by inequations stating that the selectors are downward simulations from the product bifunctor to the bifunctor that selects one of its operands

$$(p \times q) ; \pi p^{>} q^{>} \ \underline{d} \ \pi^{<} p^{<} q ; p$$

$$(p \times q) ; \pi p^{>} q^{>} \ \underline{d} \ \Pi^{<} p^{<} q ; q$$

The stronger equations, of course, remain true for a "lazy" functional language, in which no result is computed until it is known to be needed. However, this apparent optimisation usually involves some run-time overhead, which is not acceptable in a procedural language.

Selection gives a way of passing from a product type to one of its component types. We now need a method of passing from a component type to a product type. Mathematically, the easiest way of doing this is by the mirror analogue of V , which will be denoted

$$\Delta b : b \to b \times b$$

When applied to an x of type b this produces the pair (x,x) consisting just of the two copies of x . In a language without an updating assignment, this can be done very cheaply by copying pointers. In a procedural environment like that of UNIX, Δ corresponds to the fork by which parallel processes are generated. This involves copying the entire machine state. There are some things in the world that cannot be copied, for example, the world itself, and each person who lives in it. But mathematics has no concern with these practical details.

The meaning of Δ can be given (without mentioning components) by the mirror for the law for V

$$\Delta^{<}p \,;\, (p \times p) \;=\; p \,;\, \Delta p^{>}$$

The left hand side describes the construction of a pair of identical values followed by the application of p to each of them. The right hand side describes the more efficient technique of applying p to the single value before taking the copy.

But in a programming language which permits non-determinism, the effect of these two executions is not always the same. If p is non-deterministic, the two occurrences of p on the left hand side may produce different results, even when starting with the same value. However, equal results of the left hand side are still possible (by chance, say) . So the left hand side can only be inferior in the sense that it is more non-deterministic. The right hand side is still a valid optimisation, as expressed by the upward simulation property

$$p \,;\, \Delta p^{>} \subseteq \Delta^{<} p \,;\, (p \times p)$$

This means that upward simulation by itself is no longer valid in a language which permits both copying of abstract data-types and non-determinism; and total simulation has to be used.

Contravariance

Let us consider now a function h which satisfies the following distribution laws

$$hp : hp^{>} \to h^{<}p$$

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

Because distribution of h through composition reverses the order of the operands, it is known as a contravariant functor (in contrast to the normal covariant kind). The familiar converse of a relation is a contravariant functor.

The introduction of such a functor as a generator into a programming language maintains the validity of total simulation. However, the extension of (d,u) to the newly generated elements of L needs to be defined in a similar contravariant fashion

$$(d,u)(\text{"h"}b) = (h(ub), h(db))$$

Such a definition is not possible for separate upward and downward simulations, which are invalidated by a contravariant generator.

Theorem . The extended definition given above is still a retraction

Proof. d("h"b) ; u("h"b)

by contravariant distribution through generators

$$= h(ub) \,;\, h(db)$$

by contravariance of h

$$= h(db \,;\, ub)$$

(d,u) is a retraction

$= h(Gb)$

functors distribute through generators

$= G(\text{"}h\text{"}b)$

The other half of the proof is similar

end of proof

<u>Theorem .</u>Contravariant functors maintain validity of total simulation

Proof: (for the upward part)

$F(\text{"}h\text{"}p) \, ; \, u(\text{"}h\text{"}p)^{>}$

distribution through generators (contravariant for u)

$= h(fp) \, ; \, h(d^{<}p)$

contravariant distribution of h

$= h(d^{<}p \, ; \, Fp)$

by the induction hypothesis, $d : G \, \underline{d} \, F$ and monotonicity of h

$\underline{c} \; h(Gp \, ; \, dp^{>})$

by a mirror argument

$= u^{<}(\text{"}h\text{"}p) \, ; \, G(\text{"}h\text{"}p)$

end of proof

The arguments given above apply also to contravariant bifunctors. But a more interesting kind of bifunctor is one which is contravariant in one argument (the first, say) and covariant in the other

$hpq : hp^{><}q \to h^{<}pq^{>}$

h (p;q) (r;s) = hqr ; hps

The introduction of such a functor as a generator maintains validity of total simulation, provided that this is extended to distribute through "h" in a similar mixed fashion

(d,u) ("h"bc) = (h(ub)(dc),h(db)(uc))

The proofs (I hope) are a mixture of those given above.

A natural transformation between such bifunctors would satisfy the laws

nbc : hbc -> jbc

hpq ; n$^{<}$pq$^{>}$ = np$^{><}$q ; jpq

Junctional Generators

A functional generator was defined as one that admits distribution from both sides by composition. It is therefore a special case of a junction from the identity endofunctor to itself. It preserves validity of all kinds of simulation. This is a property enjoyed by all junctions.

Theorem . Generators which are junctions preserve validity of all kinds of simulation.

Proof: Let q : b -> Uc

and so Θbcq : Vb -> c

note: U(dc) = d("U"c) = dq$^{>}$.

G("Θ"bcq) ; d("Θ"bcq)

distribute functors through generators, and property 0 of "Θ"

= Θ(Gb)(Gc)(Gq) ; dc

dc : Gc -> Fc , and property 1 of Θ

$= \Theta(Gb)(Fc)(Gq\ ;\ U(dc))$

see note above

$= \Theta(Gb)(Fc)(Gq\ ;\ dq^{>})$

d is downward, monotonicity

$\subseteq\ \Theta(Gb)(Fc)(db\ ;\ Fq)$

property 1 of Θ and db : Gb -> Fb

$= V(db)\ ;\ \Theta(Fb)(Fc)(Fq)$

distribution of generators

$= d(\text{"V"}b)\ ;\ F(\text{"}\Theta\text{"}bcq)$

property 0 of "Θ"

$= d^{<}(\text{"}\Theta\text{"}bcq)\ ;\ F(\text{"}\Theta\text{"}bcq)$

end of proof.

Higher order functions

An useful example of a bifunctor of mixed variance is the one that forms from data types b and c the exponential data type (b => c) . Its values are functions from b to c , in that they take a single argument of type b and deliver a single result of type c . If p : b -> b' and q : c -> c' , then (p => q) is a function which takes as argument a function f : b' -> c , and has as its result the composed function (p;f;q) , or in standard notation (q o f o p) . This resulting function itself expects an argument of type b and gives a result of type c' . In familiar lambda-notation, the exponential can be defined as the higher order function (functional)

$$(p \Rightarrow q) = \lambda f : (b' \Rightarrow c) . (\lambda x : b . q(f(px)))$$

The mix-variant functorial property of => can be proved from this definition, by showing the equality of the two sides of the equation when applied to an arbitrary f .

Proof. ((p=>q) ; (r=>s))f

beta-substitution in the first function of the composition

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

beta-substitution in the second function

= r;p;f;q;s

definition of =>

= (r;p) => (q;s)

end of proof

Consider a function f : b x c -> a , which takes a pair of arguments. The curried version of f is the same as f , except that it takes its arguments one at a time. Thus (curry f) : b -> (c => a) is a function which expects an argument x of type b , and delivers as result another function from c to a . When this latter function is applied to an argument y in b , it delivers the same result as f does when applied to the pair (x,y) . More simply, in symbols

((curry f)x)y = f(x,y)

The currying operator has an inverse called "uncurry". Consider a function g : b -> (c=>a) . Then

uncurry g : b x c -> a

(uncurry g)(x,y) = (gx)y

It follows that

curry(uncurry g) = g

uncurry(curry f) = f

In category theory, the currying operator is represented by a new kind of junction C , with four arguments instead of three. Its defining properties are

0. Cbcaf : b -> (c => a) for f : b x c => a

1. $C^{<}p^{<}qr^{>}$((pxq) ; f ; r) = p ; $Cp^{>}q^{><}$rf ; (q=>r)

Perhaps we should check here that the lambda-definition of curry has these properties.

Theorem . The introduction of the currying operator maintains validity of total simulation.

Proof.

Note 0. by property 0 of "C" and mix-variant distribution of simulation

u("C"bcaf) = u(c=>a) = (dc => ua)

Note 1. (Fb x dc) ; u(b x c)

distribution laws for u and x

= (Fb;ub) x (dc;uc)

Fb = $^{<}$ub and (d,u) is a retraction

= (ub x Gc)

end of notes.

Consider first the upward simulation

F("C"bcaf) ; u("C"bcaf)

distribution, note 0, and introduction of identity Fb

= Fb ; C(Fb)(Fc)(Fa)(Ff) ; (dc => ua)

dc : Gc -> Fc , ud : Fd -> Gd and property 1 of C

= C(Fb)(Gc)(Ga)((Fb x dc) ; Ff ; ua)

$f^{>} = a$, $u : F \subseteq G$ and monotonicity of everything

$\subseteq$ C(Fb)(Gc)(Ga)((Fb x dc) ; u(b x c) ; Gf)

note 1 and introduction of identity Ga

= C(Fb)(Gc)(Ga)((ub x Gb) ; Gf ; Ga)

ub : Fb -> Gb and property 1 of C

= ub ; C(Gb)(Gc)(Ga)(Gf) ; (Gc => Ga)

property 0 of C and cancellation of identity

= $u^{<}$("C"bcaf) ; G("C"bcaf)

Now consider the downward simulation

G("C"bcaf) ; d("C"bcaf)

....

end of proof

Conclusion

What does it all mean? Why do the algebraic proofs work out so neatly? What is the good of it all? I should be most grateful for answers to these questions.