DATA REFINEMENT IN A CATEGORICAL SETTING

by

He Jifeng
C.A.R. Hoare

Technical Monograph PRG-PRG–90
ISBN 0-902928-68-6

November 1990

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Data Refinement in a Categorical Setting

He Jifeng and C.A.R. Hoare

August 15, 1990

Contents

1 Introduction

Data refinement is one of the most effective formal methods for design and development of large programs and systems [6, 8, 11, 12, 15]. Design starts with a grtapg D, whose nodes are names $\{b, c, .., d\}$ of basic types, and whose arrows are the names $\{p, q, .., r\}$ of primitive operations on values of the type. We introdnce the notation $\overleftarrow{p}$ to stand for the source node of p in D, and $\vec{p}$ for the target node. A program text over graph D 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 D. There are assembled into a program by means of the constructors of the language L.

An interpretation of a graph D is given by a graph morphism A, mapping each type names (objects) of D to a set of mathematical values, and each operation name of D to some mathematical function. The domain of the function Ap is just $A(\overleftarrow{p})$ and the image is included in $A(\vec{p})$, i.e., the target of Ap. This is becanse A is a graph morphism, and so respects type structure.

A large program can now be designed as an abstract program, which applies mathematical fnnctions to set of mathematical values. In the text of this program, the names $b, c, .., d$ are nsed to denote the types Ab, Ac,.. Ad; and the names $p, q, .., r$ denote the functions Ap, Aq,..Ar. Full advantage can be taken of the simplicity and generality of mathematics to ensure correctness of the design.

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

An operation with several arguments can be represented by a simple arrow from a node representing a cartesian product of the types of all the arguments; other arrows from this node to the nodes denoting types of argnments represent projections both in abstract and in concrete interpretations. A similar encoding is used in sketches [1]. So without loss of gererality, the complexity of mntiple arguments can be ignored.

But of course it is esseutial to prove that replacement of the abstract interpretation A by the concrete interpretation C maintaius the correctness of the program. In data refinement, this is done with the aid of a collection n of abstract functions, one for each type-name d of D. The function nd maps the values of tbe concrete set Cd to the corresponding abstract ste Ad. The abstraction functions are proved to commute with all the primitive operations p of D, in the following sense:

> To apply the abstract function $n\ \vec{p}$ after a concrete operation Cp gives the same result as applyiug the abstract function $n\ \overleftarrow{p}$ before the corresponding abstract operation Ap.

This description presuppose that the sets and functions over which C and A 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*. The commuting principle described in the inset above can thus be expressed by the algebraic laws:

$$\begin{aligned} \overleftarrow{nd} &= Cd \\ \overrightarrow{nd} &= Ad \\ Cp;\, n\,\overrightarrow{p} &= n\,\overleftarrow{p};\, Ap \end{aligned}$$

In category theory, the abstract and concrete interpretations are *functors* from the graph D to category M; and a collection of functions n which satisfy these laws is called a *natural transformation* from functor C to functor A. The laws are often abbreviated to the notation

$$n : C \stackrel{\cdot}{\rightarrow} A$$

or expanded to a commuting diagram

$$\begin{array}{ccc} & Ap & \\ A\,\overleftarrow{p} & \dashrightarrow & A\,\overrightarrow{p} \\ n\,\overleftarrow{p} & & n\,\overrightarrow{p} \\ C\,\overleftarrow{p} & \dashrightarrow & C\,\overrightarrow{p} \\ & Cp & \end{array}$$

In data refinement, the commuting equation is proved to hold just for the primitive operations p in D. 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 condition under which such a belief is valid. It will be shown in later section that these conditions can be expressed as algebraic laws, which must be satisfied by the constructors of the programming language L.

For this reason, the validity of data refinement depends crucially on the details of the semantics of the programming language L. Here we take an algebraic approach to semantics [4, 18], which matches well to the description of data refinement given above. The semantics is specified in three relatively independent parts; the first of these is called the *inner syntax* of the language, and the third is the *outer syntax* [18].

- A finite set of names of primitive data types, and the names of the operations that may be performed upon them; some of them map on type to another. The set is structured as a diagram D (a *direct graph*), which defines the type constraints of the context-dependent syntax of the primitive operations of the language. This diagram is similar in structure and purpose to that described above for designing concrete representations for abstract data types.

- An abstract interpretation A of the graph D is a function which maps each node name (type) of D to a mathematical set, and each arrow name (operation) of D to a function from its source set to its target set. For some languages, the arrows are mapped by A to partial functions or (in the case of non-determinism) to relations. The image of A can be made into a category M, by including all type-consistent finite compositions of the operations. Consistency of syntax with semantics requires that A should preserve the

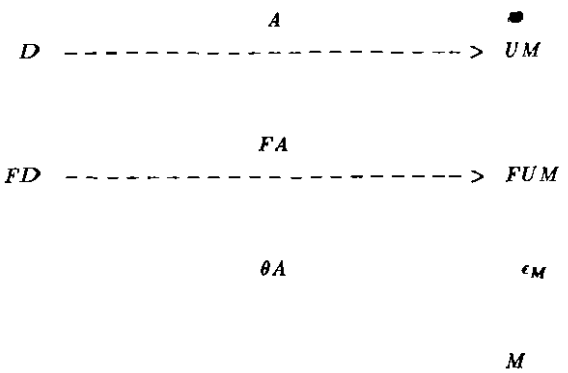
structure of the graph D; in other words, A must be a graph morphism from D to UM, where U is the forgetful functor from the variety within which M resides to the category of graphs. Later we will need its *free adjoint* F.

- The syntax and type constraints of the outer language are specified as a heterogeneous signature Σ, containing symbols for all the construtors of the language (such as sequential composition, conditional, or recursion), and sorts for its types. The set of sorts may themselves be defined by means of the type constructor (e.g., products and disjoint sums) in the set Σ, so that the number of sorts is denumerably infinite. An algebraic semantics for the language is given by a set of equations E governing the constructors in Σ. This defines a variety V, whose objects are heterogeneous (many sorted) algebras that are closed with respect to syntactically consistent application of the constructors of Σ, and satisfy the equation in E; and whose arrows, say $h : B \to C$, are Σ-homomorphisms preserving all the constructors of Σ in the sense that

$$h(\sigma_B(x_1, .., x_n)) \;=\; \sigma_C(hx_1, .., hx_n)$$

  for all constructors σ with arity n and all x_i in the carrier of B, where σ_B is the interpretation of the constructor σ in the Σ-algebga B. The variables x_i are constrained to sorts which make terms meaningful.

We define the meaning of the langnage in a manner which takes as parameter the interpretation described in (2). For a given graph D defining the inner syntax and given variety V defining the outer syntax, the language is defined as FD, the *free category* in V corresponding to D in the category of graphs. Subsequently, we may choose an object of V as the target semantic category M; and then choose a fnnction A from D to M, which is represented as a graph morphism from D to UM, as described in (2). Then the concept of *freeness* ensures that for any given interpretation A there is an unique homomorphism θA in V which maps each program of FD to its meaning in M:



where θ is the *adjunction* and ϵ the *counit*. Since V is a variety, θA is a denotational semantics in the sense that it respects the structure of the language. Thus we have preserved the greatest possible independence in the choice of V, D and finally A.

In all programming languages of interest, there exists a composition operator (denoted here as $p;q$ elsewhere $q \cdot p$). This is always associative and has both a left and right unit. It will usually be a partial operator, defined only between certain compatible sorts. There sorts play the same role as homsets in a category. To cut a long story short, the composition operator

of our programming languages will make each object of the variety V into a *small category* (usually with additional structure), and each homomorphism into a functor (which respects the additional structure); so V itself is a subcategory of CAT, the category of small categories and functors.

This method of defining a semantics for a programming language makes it possible to use data refinement as a method of establishing correctness of an implementation of the whole language. Suppose a particular choice $A : D \to M$ is made, which is nicely abstract and so easy to understand and use; but perhaps it is impossible or unacceptably inefficient to implement directly on a computer. Instead, we propose to implement the semantics associated with a more clever and complicated *concrete* interpretation $C : D \to M$. This proposal is acceptable if we can show that the two semantics are isomorphic, that is, if there exists a natural isomorphism from θC to θA. In many cases, the isomorphism may be weakened to a projection, or a natural transformation, or other even weaker form of *simulation*.

Let M be a category and let $A, C : D \to UM$ be graph morphisms, and $n : C \stackrel{\cdot}{\to} A$ a natural transformation. 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 called θn from θC to θA:

$$\theta n : \theta C \stackrel{\cdot}{\to} \theta A$$

Consider now the variety $V = CAT$. In this variety, the free category FD is just the *path category* over D, which will henceforth be denoted D^*. A simple and familiar induction shows that $n : \theta C \stackrel{\cdot}{\to} \theta A$ is also a natural transformation; and so C is a valid refinement of A. To prove this, it is sufficient to prove the commutativity property of the natural transformation just for the small set of arrows of the graph, which is a simple task compared with proving it for all programs in the language FD. In more formal terms, the adjunction θ can be extended to all such natural transformations by defining

$$\theta^+(n : C \stackrel{\cdot}{\to} A) \stackrel{def}{=} (n : C \stackrel{\cdot}{\to} A)$$

for all $C, A : D \to UM$ in $GRAPH$, i.e., the same function n (collection of arrows in M) serves as a natural transformation in both $GRAPH$ and V. We summarise this fact by stating that the variety CAT *respects* natural transformation.

Lemma 1.1
Let $f, g : D \to E$ be graph morphisms, and $n : f \stackrel{\cdot}{\to} g$ a natural transformation. Then n is also a natural transformation between functors f^* and g^* where f^* and g^* are the unique extentions of f and g on the path category D^*.
Proof: D^* and D have the same set of nodes (objects), thus the domain of n is unchanged. Consider any path $r = p_1; ..; p_k$ in the graph D, one has

$$\begin{aligned} & f^* r ; n \overrightarrow{r} \\ = \; & f p_1 ; .. ; f p_k ; n \overrightarrow{p_k} \\ = \; & n \overleftarrow{p_1} ; g p_1 ; ... g p_k \\ = \; & n \overleftarrow{r} ; g^* r \end{aligned}$$

The language FD described in the previous paragraph is a trival one, in which the only way of combining programs is by sequential composition. A more interesting and useful language will have means of constructing conditionals, loops, and structured data types. Consider, for example, a typed lambda-calculus over a graph D. This can be defined [13] as the free object over D in the variety CCC of cartesian closed category. The important question therefore arises, can the programmer in this language safely use data refinement as a program development technique ? The answer is no for natural transformations, but yes for certain other

kinds of simulation, namely Scott retractions. The proof of this requires us to extend the adjunction θ from just functors to all appropriate kinds of simulation in $GRAPH$. This will be more complicated than before, because a free cartesian closed category usually has many more objects than the graph D, and $\theta^+ n$ must be defined for those constructed objects as well. So it is far better to rely on some general theorem that guarantees the existence of $\theta^+ n$, without going to the trouble of finding it in each case. That is the technical content of the remainder of this paper.

The general technique of defining θ^+ is based on the concept of *horizontal composition* of *vertical arrows* in a *2-category*. A 2-category is a category under horizontal composition, where each of its homsets is also a category under *vertical composition* [5]. The classic example is NAT, the category of small categories, whose vertical objects are functors, and whose vertical arrows are natural transformations; in this case, horizontal and vertical compositions have their usual meanings. Other examples are obtained by restricting the arrows of the vertical category to natural isomorphisms, retractions, or other kinds of simulation. A *2-functor* is a function between 2-categories that respects both vertical and horizontal compositions and identities.

If CC is a 2-category, its *thinning* CC^- is defined as the subcategory whose objects are the horizontal objects of CC, and whose arrows are the vertical objects. If FF is a 2-functor, its *thinning* FF^- is defined as the result of thinning its source and target. A 1-functor F is said to *respect* a certain category (for example, natural transformations) if it is a thinning of a 2-functor F^+. A adjunction $< F, G, \epsilon, \delta >$ between 1-categories CC^- and DD^- is said to *respect* the 2-category CC if F does so.

The category $GRAPH$ can be seen as a thinning of the 2-category $GRAPH^+$, whose vertical objects are graph morphisms, and whose vertical arrows are simulations being used for data refinement. In a similar way the Σ-variety V can be regarded as a thinning of the 2-category V^+, whose vertical objects are Σ-homomorphisms, and whose vertical arrows are simulations. In the diagram below, we assume that the 2-functor $F^+ : GRAPH^+ \rightarrow V^+$, which maps graph morphisms to Σ-homomorphisms, and simulations to simulations, has the free functor F as its thinning. The required vertical arrow between θC and θA is provided by defining $\theta^+ n$ similarly to θC and θA as the horizontal composition of $F^+ n$ and the counit ϵ_M, which is a vertical identity in the 2-category V^+, so $\theta^+ n : \theta C \rightarrow \theta A$.

$$
\begin{array}{ccc}
 & FA & \\
FD & Fn & FUM \\
 & FC & \\
 & & \epsilon_M \quad \epsilon_M \\
 & \theta^+ n \quad \theta A & \\
 & \theta C & M
\end{array}
$$

In general, given a variety V and given one kind of simulation between graph morphisms, if we can prove

1. $GRAPH$ and V are the thinnings of the 2-category of the given kind of simulation.
2. the free functor $F : GRAPH \rightarrow V$ is respectful

then the adjunction θ between $GRAPH$ and V can be extended by defining

$$\theta^+ \stackrel{def}{=} F^+ n \fatsemi \epsilon_M$$

where $\fatsemi$ is the horizontal composition in V^+. Consequently, data refinement based on the given simulation can be used safely as a program development technique in the language with V as their outer syntax. In this case we say that the variety V (or the signature Σ together with the equation E) respects that kind of simulation.

The general way to prove respectfulness of a free functor F is by constructing the free 2-functor F^+ in a manner which mirrors the standard construction of the initial Σ-algebra. The construction is complicated by the need to keep account of the object structure, which defines the sorts of the heterogenerous Σ-algebra. Usually, there will be many more of these than in the original graph D. Consequently, FD must expressed as a colimit of the following chain in the category CAT

$$D^* \to \Phi(D) \to \Phi^2(D) \to \cdots$$

where Φ is a functor from $GRAPH$ to CAT, and $\Phi(D)$ is defined as the path category over the graph $\Sigma(D)$, where nodes are identified with formal terms $\sigma(b_1,..,b_k)$ where σ is a type constructor of arity k in the set Σ, and all b_i are nodes in D, and whose arrows are identified with formal terms $\sigma(p_1,..,p_k)$ where σ is a constructor of arity k and all p_i are arrows in D. These formal terms are considered as quotiented by equations in E. In this case, as the colimit of the chain, FD is actually the union of the categories $\Phi^i(D)$. The main theorem in section 2 shows that if Φ is respectful so is F. Thus the proof task (2) mentioned previously will he replaced by showing that the functor Φ is respectful; this greatly simplifies our job.

This paper investigates a series of constructors which enrich the simple programming language D^*, including least upper bounds, zero morphisms, coproducts, products or smash products, and higher function order spaces. Relying on the following fact (lemma 4.3)

> If Σ_1 and Σ_2 respect simulations of the kinds S_1 and Σ_2 respectively, then the signature $\Sigma_1 \cup \Sigma_2$ will respect simulations in $S_1 \cap S_2$

we can treat each enrichment separately in section 4, so that the proofs apply to the widest possible variety of languages.

The remainder of this paper is organized as follows. The next section is devoted to presenting the relevant concepts of category theory, and exploring the cocontinuity of the thinning functor. The proofs of the general theorems are postponed to the third section. Section 4 describes the concept of simulation and investigates a variety of constructors. We apply the theoretical results of section 4 to a selection of constructors in a range of familiar languages in section 5. The final section suggests a valuable criterion for the design of programming languages.

2 Preliminaries

We presume familiarity with the standard notions of category, morphisms, functor, limit, colimit and adjunction [14]. For the concept of 2-category, we refer reader to [5], but we shall not presume familiarity with it. The set of morphisms from object x to object y in category C is denoted $C(x,y)$. We will equate an object with its identity arrow, even in the case of a graph. We compose morphisms in diagram order: if $k \in C(x,y)$ and $l \in C(y,z)$ then $k;l \in C(x,z)$. We write functional application in the conventional way: if $f : B \to C$ and $g : C \to E$ are functors, and $k \in B(x,y)$, then $g(f(k)) \in E(g(f(x)), g(f(y))$.

Let B and C be small categories, f, g and h be functors from B to C. Let $m : f \stackrel{\cdot}{\to} g$ and $n : g \stackrel{\cdot}{\to} h$ be natural transformations

$$\begin{array}{ccc} & \xrightarrow{\quad h \quad} & \\ & n & \\ B & \xrightarrow{\quad g \quad} & C \\ & m & \\ & \xrightarrow{\quad f \quad} & \end{array}$$

The vertical composition $(m;n) : f \stackrel{\cdot}{\to} h$ is defined by

$$m;n \stackrel{def}{=} \lambda b.(mb;nb)$$

Note that functor f is a vertical object, and the associated identity arrow is $f_0 : f \stackrel{\cdot}{\to} f$ where f_0 is the object function of f. $\uparrow$ and $\downarrow$ will denote the target function and source function in the vertical composition.

Given functors and natural transformations as below

$$\begin{array}{ccccc} & \xrightarrow{\quad h \quad} & & \xrightarrow{\quad h' \quad} & \\ & n & & n' & \\ B & \xrightarrow{\quad g \quad} & C & \xrightarrow{\quad g' \quad} & D \\ & m & & m' & \\ & \xrightarrow{\quad f \quad} & & \xrightarrow{\quad f' \quad} & \end{array}$$

the horizontal composite $(m \mathbin{;} m') : (f;f') \stackrel{\cdot}{\to} (g;g')$ is defined by

$$m \mathbin{;} m' \stackrel{def}{=} \lambda b.(m'(fb);g'(mb))$$

We now summarise a few familiar facts about NAT. If $ID(C) : C \to C$ is the identity functor for the category C, and its restriction to objects $Id(C)_0 : Id(C) \stackrel{\cdot}{\to} Id(C)$ is the identity natural transformation of that functor, one has

$$\begin{aligned} m \mathbin{;} Id(C)_0 &= m \\ Id(C)_0 \mathbin{;} m &= m \end{aligned}$$

Each category C will be called a *horizontal object*, and $Id(C)_0$ will be called the *horizontal identity* in the 2-category NAT [14].

The composition ; and $\mathbin{;}$ in NAT are readily seen to be associative. Moreover, they are related by the interchange law

$$(m;n) \mathbin{;} (m';n') = (m \mathbin{;} m');(n \mathbin{;} n')$$

Let CC and DD be 2-categories. A 2-functor $ff : CC \to DD$ sends objects of CC to objects of DD, arrows of CC to arrows of DD, preserving source and target and all types of identity and composition. A 2-natural transformation $\beta : ff \stackrel{\cdot}{\to} gg : CC \to DD$ assigns to each horizontal object x of CC an arrow $\beta(x) : ff(x) \to gg(x)$ in DD, which is not only natural in the ordinary sense that, for any vertical object $p : x \to y$, we have

$$ff(p) ; \beta(y) = \beta(x) ; gg(p)$$

but also 2-natural in the sense that, for each vertical arrow $m : p \stackrel{\cdot}{\to} q$ where $p, q \in CC(x, y)$, we have

$$ff(m) \fatsemi \beta(y) = \beta(x) \fatsemi gg(m)$$

We will use EE to denote the 2-category with horizontal objects all small 2-categories, and vertical objects all 2-functors and arrows all 2-natural transformations between them. We define a thinning functor from EE to NAT, and show that this preserves the colimit of the right chain in EE. This is needed to ensure that the thinning of a free 2-adjunction is a free adjunction.

If CC is a small 2-category (an object in EE), its thinning CC^- is defined as the subcategory whose objects are horizontal objects of CC, and whose arrows are the vertical objects. If $ff : CC \to DD$ is a 2-functor (a vertical object in EE), its thinning ff^- is defined as the result of thinning its source and target. Similarly, a 2-natural transformation is mapped to an ordinary one in NAT. It is clear that $ff^- : CC^- \to DD^-$ is a functor. A functor $f : CC^- \to DD^-$ is said to be respectful if it is the thinning of a 2-functor $f^+ : CC \to DD$.

Let $hh : NAT \to NAT$ he a 2-functor. Suppose that a family of inclusion functors $\{incl(C) : C \to hh(C) \mid C is a small category\}$ is a 2-natural transformation from the identity functor to hh. Consequently, for any small category C there is a right chain in NAT

$$C \xrightarrow{in_0} hh(C) \xrightarrow{in_1} hh^2(C) \xrightarrow{in_2} \cdots$$

where $in_i \stackrel{def}{=} hh^i(incl(C))$ is an inclusion functor. In this case, the colimit of the chain, denoted by $Lim_i hh^i(C)$, is actually the union of the categories $hh^i(C)$

$$Lim_i hh^i(C) = \bigcup_i hh^i(C)$$

For any functor $f, g : C \to D$ and any natural transformation $n : f \stackrel{\cdot}{\to} g$, one has the following commuting diagram

$$\begin{array}{ccccccc}
 & & & & & & Lim_i hh^i(C) \\
 & C & - - - - > & hh(C) & - - - - > & hh^2(C) & - - - - > \\
n & & & hh(n) & & hh^2(n) & \\
 & D & - - - - > & hh(D) & - - - - > & hh^2(D) & - - - - > \\
 & & & & & & Lim_i hh^i(D)
\end{array}$$

We define $Lim_i hh^i(n)$ as the mediating arrow from the colimit $Lim_i hh^i(C)$ to the cocone $Lim_i hh^i(D)$, i.e., for all $k \geq 0$

$$hh^k(n) \fatsemi t_{D_k} = t_{C_k} \fatsemi Lim_i hh^i(n)$$

where $t_{C_k} : hh^k(C) \to Lim_i hh^i(C)$ and $t_{D_k} : hh^k(D) \to Lim_i hh^i(D)$ are inclusion functors. It is easy to show that $Lim_i hh^i$ is a 2-functor defined on NAT

When $h : CAT \to CAT$ is a functor, and the family of inclusion functors $incl_C : C \to h(C)$ is a natural transformation from the identity functor to h, the colimit $Lim_i h^i$ can be defined in the same way as $Lim_i hh^i$.

The following theorem tells us that the thinning operator is a cocontinuous functor.
Theorem 2.1 $Lim_i hh^i = Lim_i (hh^-)^i$
Proof: Direct from the definition of the thinning operator.

3 Data Refinement By Natural Transformation

A graph of primitive types and operations would be a very primitive language, which offers no method at all of combining built-in operations into useful programs. The introduction of a constructor denoting composition greatly increases the power of the language, since it permits operations to be assembled into sequences. We will use Σ to denote an arbitrary set of constructors one of which is composition. Without loss of generality, we also assume that the identity constructor is in Σ. If σ is a constructor (binary, say), then $\sigma(x,y)$ is taken as the text formed from texts x and y separately by commas, prefixed by σ and open bracket, and terminated by close bracket.

A Σ-category is defined as a small heterogenerous Σ-algebra. One of its sorts is the set of all its objects. The remaining sorts are identified with a homset, i.e., a pair of objects. Each object is either a node in the graph or built from nodes by the means of the type constructors in Σ. All constructors in Σ 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 of a Σ-category is defined in the usual way as the smallest set containing all constants denoting each object and arrow in the graph, and closed with respect to syntactically consistent application of the constructors of Σ. For a given Σ-category M and any constructor σ of Σ, σ_M will stand for the interpretation of σ in M.

Let M and N be Σ-categories. A Σ-homomorphism $h : M \rightarrow N$ is a function on the carrier sets which preserves all the constructors of Σ in the sense that

$$h\sigma_M(x_1,..,x_k) = \sigma_N(hx_1,..,hx_n)$$

for all $\sigma \in \Sigma$ and x_i in the carrier of M. Since the composition is included in Σ, a Σ-homomorphism also preserves composition, and therefore is a functor in the usual sense. We shall be particularly interested in cases where σ_M is a function on M, whose defining properties are expressed by categorical concepts; for example, it may be an endorfunctor, or a natural transformation. The variables x_i are (implicitly here) constrained to sorts which makes these terms meaningful.

A Σ-variety is defined as a category V whose objects themselves Σ-categories, and whose arrows are Σ-homomorphisms. Since Σ-homomorphisms are functors, a Σ-variety is a subcategory of CAT. A Σ-variety is nsually defined by a set of equational laws governing the constructors of Σ, and its objects are just those Σ-categories in which the laws are valid. CAT itself is a Σ-variety, with composition and identity as the only members of Σ, and familiar axioms of category theory as equations. Cartesian closed categories form another variety, with product and exponential endofunctors serving both as sort constructors and operators of Σ.

Let Σ be a set of constructors in a programming language. A representation of Σ is a pair (Φ,H), with Φ a functor from $GRAPH$ to CAT, and for all Σ-categories M, H_M a functor from $\Phi(UM)$ to M where U is a forgetful functor from the variety V to the category $GRAPH$, such that for all graphs D,E and graph morphisms $f : D \rightarrow E$,

1. $\Phi(D)$ is the path category over the graph $\Sigma(D)$, whose nodes are identified by $\sigma(b_1,..,b_k)$ where σ is a type constructor of arity k in the set Σ, and all b_i are nodes in D, and whose arrows are identified by $\sigma(p_1,..,p_k)$ where σ is a constructor of arity k and all p_i are arrows in D.
2. $(\Phi f)\sigma(p_1,..,p_k) = \sigma(fp_1,..,fp_k)$ for $\sigma \in \Sigma$ and $p_i \in D$.
3. $\sigma_M(p_1,..,p_k) = H_M(\sigma(p_1,..,p_k))$ for $\sigma \in \Sigma$ and $p_i \in M$

Σ is said to be *representable* if such a representation exists. In this case Φ is called a *representation functor* of Σ. Since Σ contains the identity constructor, there is a right chain for any graph D

$$D^* \xrightarrow{in_0} \Phi(D) \xrightarrow{in_1} \Phi^2(D) \xrightarrow{in_2} ..$$

where $in_i \stackrel{def}{=} \Phi^i(incl(D))$ and $incl(D) : D^* \to \Phi(D)$ are inclusion functors.

Σ is said to be *respectable* if the representation Φ is respectful.

Let (Φ, H) he a respectful representation of Σ. Define

$$\begin{aligned} F &\stackrel{def}{=} Lim_i \Phi^i \\ \delta &\stackrel{def}{=} \lambda D.t_D \\ \epsilon_0(M) &\stackrel{def}{=} Id(M) \\ \epsilon_{k+1}(M) &\stackrel{def}{=} \Phi(\epsilon_k(M)); H_M \qquad for\ k \geq 0 \end{aligned}$$

where t_D is the inclusion functor from D to $UF(D)$.
Lemma 3.1 $\{\epsilon_k(M) \mid k \geq 0\}$ is a cocone of the right chain

$$M \xrightarrow{in_0} \Phi(UM) \xrightarrow{in_1} \Phi^2(UM) \xrightarrow{in_2} \cdots$$

where $in_i \stackrel{def}{=} \Phi^i(incl(M))$ and $incl(M) : M \to \Phi(UM)$ are inclusion functors.
Proof: From the definition of H_M it follows that for all $p \in M$

$$\begin{aligned} p &= H_M p \\ &= (incl(M); H_M)p \end{aligned}$$

which implies $\epsilon_0(M) = in_0; \epsilon_1(M)$.
Assume that $\epsilon_k = in_k; \epsilon_{k+1}(M)$, one has

$$\begin{aligned} \epsilon_{k+1}(M) &= \Phi(\epsilon_k(M)); H_M \\ &= \Phi(in_k; \epsilon_{k+1}(M)); H_M \\ &= \Phi(in_k); \Phi(\epsilon_{k+1}(M)); H_M \\ &= in_{k+1}; \epsilon_{k+2}(M) \end{aligned}$$

This completes the proof.

Define

$$\epsilon \stackrel{def}{=} \lambda M.[\epsilon_0, \epsilon_1, ..]$$

where $[\epsilon_0, \epsilon_1, ..]$ is the mediating morphism from the colimit $Lim_i \Phi^i(UM)$ to the cocone $\{\epsilon_k \mid k \geq 0\}$ satisfying for all inclusions $t_{M_k} : \Phi^k(UM) \to Lim_i \Phi^i(UM)$

$$t_{M_k}; [\epsilon_0(M), \epsilon_1(M), ..] = \epsilon_k(M)$$

Now we are going to show that $< F, U, \delta, \epsilon >$ is a respectful adjunction.
Lemma 3.2 F is a respectful functor from $GRAPH$ to the variety V.
Proof: Here we first wish to prove that for any graph morphism $f : D \to E$, Ff is a Σ-homomorphism. Let σ he a constructor of arity k, and $p_1, .., p_k$ are arrows in D

$$\begin{aligned} & Ff(\sigma(p_1, .., p_k)) \\ = \ & \Phi(f)(\sigma(p_1, .., p_k)) \\ = \ & \sigma(fp_1, .., fp_k) \\ = \ & \sigma(Ffp_1, .., Ffp_k) \end{aligned}$$

When $p_1, .., p_k$ are elements in $\Phi^i(D)$, one has

$$\begin{aligned} & Ff(\sigma(p_1, .., p_k)) \\ = \ & \Phi^{i+1}(f)(\sigma(p_1, .., p_k)) \\ = \ & \sigma(\Phi^i(f)p_1, .., \Phi^i(f)p_k) \\ = \ & \sigma(Fp_1, .., Fp_k) \end{aligned}$$

From the definition of F and theorem 2.1 one concldes that F is respectful.

Lemma 3.3 $\epsilon(M) : FU(M) \rightarrow M$ is a functor.
Proof: Since all $\epsilon_k(M)$ are functors, so they are the vertical identity in the 2-category NAT. The conclusion follows from the fact [14] that the horizontal colimit of vertical identities is a vertical identity.

Lemma 3.4 $\delta : Id \rightarrow UF$ is a natural transformation.
Proof: For any $f : D \rightarrow UM$ one has

$$\begin{aligned} & f;\delta \overrightarrow{f} \\ = \ & f;t_{UM} && \{f : D \rightarrow UM \ and \ def \ of \ \delta\} \\ = \ & t_D;UFf && \{def \ of \ F\} \\ = \ & \delta \overleftarrow{f};UFf && \{f : D \rightarrow UM \ and \ def \ of \ \delta\} \end{aligned}$$

Lemma 3.5 $\delta(UM);U\epsilon(M) = Id(UM)$.
Proof:

$$\begin{aligned} & LHS \\ = \ & t_{UM};[\epsilon_0(M), \epsilon_1(M), ...] && \{def \ of \ \delta \ and \ \epsilon\} \\ = \ & \epsilon_0(M) && \{def \ of \ [f, g, ..]\} \\ = \ & RHS && \{def \ of \ \epsilon_0(M)\} \end{aligned}$$

For any graph morphism $f : D \rightarrow UM$ define

$$\theta f \stackrel{def}{=} Ff;\epsilon(M)$$

Lemma 3.6 θ is an injection.
Proof: Here we want to show that for all graph morphisms $f : D \rightarrow UM$

$$f = \delta \overleftarrow{f};U(\theta f)$$

This can be shown as follows

$$\begin{aligned} & \delta \overleftarrow{f};U(\theta f) \\ = \ & \delta \overleftarrow{f};UFf;U\epsilon(M) && \{def \ of \ \theta\} \\ = \ & f;\delta \overrightarrow{f};U\epsilon(M) && \{lemma \ 3.4\} \\ = \ & f;\delta(UM);U\epsilon(M) && \{f : D \rightarrow UM\} \\ = \ & f && \{lemma \ 3.5\} \end{aligned}$$

Now it is easy to see that $\theta f = \theta g$ implies $f = g$.

Theorem 3.1 $< F, U, \delta, \epsilon >$ is a respectful adjunction provided Φ is respectful.

Proof: We want to show that for any Σ-morphism $h : FD \to M$, there exists $f : D \to UM$ such that

$$h = \theta f$$

Define $f \stackrel{def}{=} t_D; Uh$. Then we wish to show that

$$\Phi^i(f); \epsilon_i = t_i; h$$

where $\epsilon_i : \Phi^i(UM) \to M$ abbreviates $\epsilon_i(M)$, and $t_i : \Phi^i D \to FD$ stands for the inclusion from $\Phi^i D$ to FD. When $i = 0$ one has

$$\begin{array}{lll} & LHS & \\ = & f; Id(UM) & \{def\ of\ \epsilon_0(M)\} \\ = & t_D; Uh & \{def\ of\ f\} \\ = & RHS & \{t_0 = t_D\} \end{array}$$

Proceeding inductively, for any constructor σ of arity k and all $p_1, .., p_k \in \Phi^n(D)$

$$\begin{array}{lll} & h\sigma(p_1, .., p_k) & \\ = & \sigma_M(hp_1, .., hp_k) & \{h\ is\ a\ \Sigma - homomorphism\} \\ = & H_M\sigma(hp_1, .., hp_k) & \{def\ of\ H_M\} \\ = & H_M\sigma((\Phi^n(f); \epsilon_n)p_1, .(\Phi^n; \epsilon_n)p_k) & \{induction\ hypothesis\} \\ = & H_M\Phi(\Phi^n(f); \epsilon_n)\sigma(p_1, .., p_k) & \{defining\ property\ of\ \Phi\} \\ = & (\Phi(\epsilon_n); H_M)(\Phi^{n+1}(f)\sigma(p_1, .., p_k)) & \{\Phi\ is\ a\ functor\} \\ = & (\Phi^{n+1}(f); \epsilon_{n+1})\sigma(p_1, .., p_k) & \{\epsilon_{n+1} \stackrel{def}{=} \Phi(\epsilon_n); H_M\} \end{array}$$

So we deduce

$$\Phi^{n+1}(f); \epsilon_{n+1} = t_{n+1}; h$$

On the other hand, we know that for all $i \geq 0$

$$\begin{array}{lll} & t_i; \theta f & \\ = & t_i; (Ff; \epsilon(M)) & \{def\ of\ \theta\} \\ = & t_i; Lim_i\Phi^i(f); \epsilon(M) & \{def\ of\ F\} \\ = & \Phi^i(f); t_i; \epsilon(M) & \{def\ of\ Lim_i\Phi^i(f)\} \\ = & \Phi^i(f); \epsilon_i(M) & \{def\ of\ \epsilon(M)\} \\ = & \Phi^i(f); \epsilon_i & \{def\ of\ \epsilon_i\} \end{array}$$

By the universal property of the colimit it follows that

$$h = \theta f$$

Furthermore, from lemma 3.6 we conclude that θ is a bijection.

For any natural transformation $n : f \to g$, define

$$\begin{array}{lll} F^+ & \stackrel{def}{=} & Lim_i(\Phi^+)^i \\ \theta^+ n & \stackrel{def}{=} & F^+ n \fatsemi \epsilon(M)_0 \end{array}$$

where Φ^+ is a 2-functor whose existence is postulated by the respectfulness of Φ, and $\epsilon(M)_0 : \epsilon(M) \to \epsilon(M)$ is the identity natural transformation on $\epsilon(M)$.

Theorem 3.2 $\theta^+ n : \theta f \stackrel{\cdot}{\rightarrow} \theta g$
Proof: From the definition of the vertical composition in the 2-category NAT, one has

$$\begin{aligned}
& \uparrow \theta^+ n \\
= \;& \uparrow F^+ n; \uparrow \epsilon(M)_0 && \{def\ of\ \stackrel{\cdot}{;}\ in\ NAT\} \\
= \;& F^+(\uparrow n); \uparrow \epsilon(M)_0 && \{F^+\ is\ a\ 2-functor\} \\
= \;& F^+ g; \epsilon(M) && \{def\ of\ n\ and\ \epsilon(M)_0\} \\
= \;& Fg; \epsilon(M) && \{F\ is\ the\ thinning\ of\ F^+\} \\
= \;& \theta g && \{def\ of\ \theta\}
\end{aligned}$$

In a similar way one can show that $\downarrow \theta^+ n = \theta f$.

In the remainder of this section we will deal with natural transformations. It will be shown that they are respectful; i.e., they have a respectful representation (Φ, H).

Let $\hat{m} : \hat{h}_1 \stackrel{\cdot}{\rightarrow} \hat{h}_2$ be a natural transformation constructor, where $\hat{h}_1$ and $\hat{h}_2$ are endofunctorial constructors, and their meanings are specified by covariant endofunctor h_1 and h_2 respectively. The interpretation of $\hat{m}$ is given by a natural transformation $m : h_1 \stackrel{\cdot}{\rightarrow} h_2$. Suppose that all the constructors but the composition in Σ are endofunctors or natural transformations between them. Then for any graph D, $\Sigma(D)$ will be the graph with each node identified by $h(b_1, .., b_k)$ where h is an endofunctorial constructor of arity k in Σ, and $b_1, .., b_k$ are all nodes in graph D, and with each arrow identified by $\sigma(p_1, .., p_k)$ where σ is a constructor of arity k in Σ and all $p_1, .., p_k$ all are arrows in D. The target and source function in the path category over $\Sigma(D)$ is defined as usual, for example, one has

$$\begin{aligned}
(\overrightarrow{mb}) &= h_2 b \\
(\overleftarrow{mb}) &= h_1 b \\
(\overrightarrow{hp}) &= h(\overrightarrow{p}) \\
(\overleftarrow{hp}) &= h(\overleftarrow{p})
\end{aligned}$$

Finally comes the main result of this section.
Theorem 3.3 The set of natural transformation constructors between covariant functors has a respectful representation (Φ, H).
Proof: For any graph morphisms $f, g : D \rightarrow E$ and any natural transformation $n : f \stackrel{\cdot}{\rightarrow} g$, define

$$\Phi^+(n)h(b_1, .., b_k) \stackrel{def}{=} h(nb_1, .., nb_k)$$

Consider three cases:
(1) σ is a covariant endofunctorial constructor h

$$\begin{aligned}
& \Phi(f)h(p_1, .., p_k); \Phi^+(n)\, h(\overrightarrow{p_1, .., p_k}) \\
= \;& h(fp_1, .., fp_k); \Phi^+(n)h(\overrightarrow{p_1}, .., \overrightarrow{p_k}) && \{def\ of\ \Phi\} \\
= \;& h(fp_1, .., fp_k); h(n\ \overrightarrow{p_1}, .., n\ \overrightarrow{p_k}) && \{def\ of\ \Phi^+\} \\
= \;& h(fp_1; n\ \overrightarrow{p_1}, .., fp_k; \overrightarrow{p_k}) && \{h\ is\ a\ functor\} \\
= \;& h(n\ \overleftarrow{p_1}; gp_1, .., n\ \overleftarrow{p_k}; gp_k) && \{n : f \stackrel{\cdot}{\rightarrow} g\} \\
= \;& \Phi^+(n)\, h(\overleftarrow{p_1, .., p_k}); \Phi(g)h(p_1, .., p_k) && \{by\ a\ mirror\ argument\}
\end{aligned}$$

(2) σ is a natural transformation $m : h_1 \stackrel{\cdot}{\rightarrow} h_2$

$$\Phi(f)m(b_1, .., b_k); \Phi^+(n)\, m(\overrightarrow{b_1, .., b_k})$$

$$
\begin{aligned}
&= m(fb_1,..,fb_k);\Phi^+(n)h_2(b_1,..,b_k) && \{def\ of\ \Phi\}\\
&= m(\overleftarrow{nb_1},..,\overleftarrow{nb_k});h_2(b_1,..,b_k) && \{def\ of\ \Phi^+\}\\
&= h_2(nb_1,..,nb_2);m(\overrightarrow{nb_1},..,\overrightarrow{nb_k}) && \{m : h_1 \stackrel{\cdot}{\rightarrow} h_2\}\\
&= \Phi^+(n)\ m(\overleftarrow{b_1,..,b_k});\Phi(g)m(b_1,..,b_k) && \{by\ a\ mirror\ argument\}
\end{aligned}
$$

(3) σ is the composition constructor ;

$$
\begin{aligned}
&\Phi(f)(p_1;p_2);\Phi^+(n)\,(\overrightarrow{p_1;p_2})\\
&= (fp_1;fp_2);n\ \overrightarrow{p_2} && \{def\ of\ \Phi\}\\
&= fp_1;n\ \overleftarrow{p_2};gp_2 && \{n : f \stackrel{\cdot}{\rightarrow} g\}\\
&= n\ \overleftarrow{p_1};gp_1;gp_2 && \{n : f \stackrel{\cdot}{\rightarrow} g\}\\
&= \Phi^+(n)(\overleftarrow{p_1;p_2});\Phi(g)(p_1;p_2) && \{def\ of\ \Phi\}
\end{aligned}
$$

So $\Phi^+(n)$ is really a natural transformation from $\Phi(f)$ to $\Phi(g)$. It is routine to check that Φ^+ respects horizontal and vertical compositions of natural transformations. Therefore Φ is the thinning of 2-functor Φ^+.

4 Data Refinement by Simulation

In program development, it is not necessary to insist on absolute identity of the effect 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 preorder $\sqsubseteq$ into the homsets of the matbematical categories, to denote its right operand is an improvement on the left operand (which must have the same domain and codomain). In the mathematical theory, $\sqsubseteq$ is an arbitrary preorder, and may be interpretated as any kind of improvement. To ensure that the improvement is maintained in all contexts, we postulate that all operators, constructors and functors under consideration are monotonic. As nsual, $\equiv$ denotes the equivalence induced by the preorder.

Let M be a small category in which for each pair (b,c) of objects $M(b,c)$ is a preorder, and moreover the composition ; is monotonic. Define MM as the collection of all pairs (p,q) of elements p,q of M with $p \sqsubseteq q$. It is well known that MM is a 2-category. The horizontal composition $\overset{\cdot}{;}$ in MM is defined by

$$(p,q) \overset{\cdot}{;} (r,s) \stackrel{def}{=} (p;r,q;s)$$

provided that $\vec{p} = \vec{q} = \overleftarrow{r} = \overleftarrow{s}$ in M.

The vertical composition ; is defined by

$$(p,q);(q,r) \stackrel{def}{=} (p,r)$$

Both definitions are valid since $\sqsubseteq$ is a preorder and the composition ; in M is monotonic. The compositions ; and $\overset{\cdot}{;}$ are readily to be associative. Moreover, they are related by the interchange law

$$((p,q);(q,r)) \overset{\cdot}{;} ((p',q');(q',r')) \;=\; ((p,q) \overset{\cdot}{;} (p',q'));((q,r) \overset{\cdot}{;} (q',r'))$$

Now the commuting equation defining natuality 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.

Let $f,g : L \to M$ be functors, an *up-simulation* u is defined as a transformation from f to g such that

$$\begin{aligned} ub &: fb \to gb && \textit{for all objects } b \textit{ in } L \\ u\overleftarrow{p};gp &\sqsubseteq fp;u\vec{p} && \textit{for all elements } p \textit{ in } L \end{aligned}$$

It will be denoterd $u : f < g$.

A *down-simulation* d is defined as a transformation from g to f such that

$$\begin{aligned} db &: gb \to fb && \textit{for all objects } b \textit{ in } L \\ gp;d\vec{p} &\sqsubseteq d\overleftarrow{p};fp && \textit{for all elements } p \textit{ in } L \end{aligned}$$

It will be denoted $d : g > f$. clearly, a natural transformation $n : f \dot{\to} g$ is both an up-simulation and a down-simulation from f to g.

One way of combining the two definitions is in the definition of a *total simulation*. this is a pair (d,u), where

2. d is a down-simulation from g to f.

3. $db; ub = gb$ and $ub; db \sqsubseteq fb$ for all objects b in L.

It will be denoted $(d, u) : f \to g$.

The following lemma shows that each component of a total simulation uniquely determines the other, up to equivalence.

Lemma 4.1 Let (d, u) and (d', u') be total simulations from f to g. Then

$$d \equiv d' \ \ iff \ \ u \equiv u'$$

Proof: Assume $d \equiv d'$. For any object b one has

$$\begin{array}{lll} & ub & \\ = & ub; gb & \\ = & ub; (d'b; u'b) & \\ = & ub; db; u'b & \\ \sqsubseteq & fb; u'b & \\ = & u'b & \end{array}$$

The proof that $u'b \sqsubseteq ub$ is similar. The proof of the reverse implication is similarly similar.

Let $(d, u) : f \to g$ and $(e, v) : g \to h$ be total simulations. Define

$$(d, u); (e, v) \stackrel{def}{=} (e; d, u; v)$$

where $e; d \stackrel{def}{=} \lambda b.(eb; db)$ and $u; v \stackrel{def}{=} \lambda c.(uc; vc)$.

Lemma 4.2 $(e; d,\ u; v)$ is a total simulation from f to h.

Proof: For all elements p in L one has

$$\begin{array}{lll} & hp; (e; d)\ \overrightarrow{p} & \\ = & hp; e\ \overrightarrow{p}; d\ \overrightarrow{p} & \{def\ of\ d; u\} \\ \sqsubseteq & e\ \overleftarrow{p}; gp; d\ \overrightarrow{p} & \{e : h > g\} \\ \sqsubseteq & e\ \overleftarrow{p}; d\ \overleftarrow{p}; fp & \{d : g > f\} \\ = & (e; d)\ \overleftarrow{p}; fp & \{def\ of\ e; d\} \end{array}$$

In the similar way we can prove

$$(u; v)\ \overleftarrow{p}; hp \sqsubseteq fp; (u; v)\ \overrightarrow{p}$$

Moreover we have

$$\begin{array}{lll} & (e; d)d; (u; v)b & \\ = & eb; db; ub; vb & \{def\ of\ u; v\ and\ e; d\} \\ = & eb; gb; vb & \{db; ub = gb\} \\ = & eb; vb & \{gb\ is\ an\ identity\} \\ = & hb & \{eb; vb = hb\} \end{array}$$

$$
\begin{array}{lll}
 & (u;v)b;(e;d)b & \\
= & ub;vb;eb;db & \{def\ of\ u;v\ and\ e;d\} \\
\sqsubseteq & ub;gb;db & \{vb;eb \sqsubseteq gb\ and\ ;\ is\ monotonic\} \\
= & ub;db & \{gb\ is\ an\ identity\} \\
\sqsubseteq & fb & \{ub;db \sqsubseteq fb\}
\end{array}
$$

So $(e;d,\ u;v)$ is a total simulation from f to h.

Given functors and total simulations as below

$$
\begin{array}{ccccc}
 & \text{- - - - - - } h \text{ - - - - - - >} & & \text{- - - - - - } h' \text{ - - - - - - >} & \\
L & (d,\ u) & M & (d',\ u') & N \\
 & \text{- - - - - - } f \text{ - - - - - - >} & & \text{- - - - - - } f' \text{ - - - - - - >} &
\end{array}
$$

the horizontal composition $(d,u) \overset{\bullet}{;} (d',u')$ is defined by

$$(d,u) \overset{\bullet}{;} (d',u') \overset{def}{=} (\lambda b.h'(db);d'(fb),\ \lambda b.u'(fb);h'(ub))$$

Lemma 4.3 $(d,u) \overset{\bullet}{;} (d',u')$ cis a total simulation from $f;f'$ to $h;h'$.
Proof: Similar to lemma 4.2.

Because neither the collection of total simulations nor the collection of up(down)-simulations satisfies the interchange law, they are not 2-categories. Therefore we need to modify the results in the previous section to take this into account: we develop a theory of *quasi 2-category* which characterises the mathematical properties of simulations. By a quasi 2-category is meant a collection of arrows with two different compositions $\overset{\bullet}{;}$ and ;, in which every identity arrow for the first composition is also an identity for the second composition. The interchange law is weakened to an inequation. From the previous lemmas we know that $SIMU$, the collection of all total simulations, is a quasi 2-category. It is also obvious that the collection of down-simulations and the collection of up-simulations are quasi 2-categories as well.

Let QC and QD be quasi 2-categories. A *quasi 2-functor* $qf : QC \to QD$ sends objects of QC to objects of QD, arrows of QC to arrows of QD, preserving source and target and all types of identity and composition, In a similar way to section 3 we can define the thinning category QC^{-}, and the thinning functor qf^{-}. A functor $f : QC^{-} \to QD^{-}$ is said to be respectful if it is the thinning of a quasi 2-functor $f^{+} : QC \to QD$. If QC and QD are categories of total simulations (down-simulations, up-simulations) f is said to respect total simulations (down-simulations, up-simulations). It is easy to see that theorem 2.1 still holds in the case of quasi 2-categories.

A Σ-*preordered category* is a Σ-category whose homsets are endowed with preorders, and all operators in Σ are monotonic. The equations E may include inequations as well as equations. As for natural transformation constructors, we can define up-simulation and down-simulation and total simulation coustructors in a Σ-preordered category N by the inequations that they satisfy

$$
\begin{array}{rl}
g_N(p);d_N\ \overrightarrow{p} \sqsubseteq_N\ d_N\ \overleftarrow{p};h_N(p) & \text{for all elements } p \in N \\
u_N\ \overleftarrow{p};g_N(p) \sqsubseteq_N\ h_N(p);u\ \overrightarrow{p} & \text{for all elements } p \in N \\
d_N b;u_N b \equiv_N\ g_N(b) & \text{for all objects } b \text{ of } N \\
u_N b;d_N b \sqsubseteq_N\ f_N(b) & \text{for all objects } b \text{ of } N
\end{array}
$$

where d_N and u_N are the interpretations of d and u in N, and $\sqsubseteq_N$ is the preorder defined on the homsets of N.

A Σ-*preordered variety* is defined as a category V with Σ-preordered categories as its objects, and with monotonic Σ-homomorphisms as its arrows. Since Σ-homomorphisms are functors, a Σ-variety is a subcategory of CAT.

Σ is said to respect total simulations (down-simulations, up-simulations) if its representation functor Φ does so, i.e., Φ is a thinning of quasi 2-functor Φ^+ defined on the qnasi 2-category of total simulations (down-simulations, up-simulations). In this case, following the same approach presented in the previous section we can construct a respectful adjunction $< F, U, \delta, \epsilon >$. It indicates that the introduction of constructors of Σ into a programming language will maintain validity of total simulation (down-simulation, up-simulation) as a data refinement rule.

In the rest of this section we are going to investigate a number of language constructors and work out their respectful representation. The following lemma enables us to treat them individually so that the results apply to various kinds of language as far as the validity of simulations in data refinement is concerned.

Lemma 4.4 If Σ_1 and Σ_2 respect simulations of the kinds S_1 and S_2 respectively, then the signature $\Sigma_1 \cup \Sigma_2$ respects simulations in $S_1 \cap S_2$.
Proof: Let (Φ_1, H_1) and (Φ_2, H_2) he the respectful representations of Σ_1 and Σ_2 respectively. It is obvious that the union set $\Sigma_1 \cup \Sigma_2$ is also representable by the pair (Φ, H) defined as follows

$$\begin{aligned}
\Phi(D) &\stackrel{def}{=} (\Sigma_1(D) \cup \Sigma_2(D))^* \\
\Phi(f)\sigma(p_1,..,p_k) &\stackrel{def}{=} \Phi_i(f)\sigma(p_1,..,p_k) \qquad if\ \sigma \in \Sigma_i \\
H_M(\sigma(p_1,..,p_k) &\stackrel{def}{=} (H_i)_M(\sigma(p_1,..,p_k)) \quad if\ \sigma \in \Sigma_i
\end{aligned}$$

Define for any simulation $t \in S_1 \cup S_2$

$$\Phi^+(t)b \stackrel{def}{=} \Phi_i^+(t)b \qquad if\ b \in \Sigma_i(D)$$

Because both Σ_1 and Σ_2 respect simulation t, so $\Phi_1^+(t)$ and Φ_2^+ are the extention s of the simulation t in the categories $\Phi_1(D)$ and $\Phi_2(D)$ correspondingly. From the definition of Φ^+ it follows that $\Phi^+(t)$ is the extention of t in the category $(\Sigma_1(D) \cup \Sigma_2(D))^*$.

Lemma 4.5 If Σ_1 and Σ_2 respect simulations of the kinds S_1 and S_2 respectively, then the siguature $\Sigma_1 \cap \Sigma_2$ respects simulations in $S_1 \cup S_2$.
Proof: Dual to lemma 4.4.

Similar to a natural transformation, a down-simulation (or an up-simulation or a total simulation) defined on graph $(D, \sqsubseteq_D)$ can he seen as a down-simulation (or an up-simulation or a total simulation) on the path category $(D^*, \sqsubseteq_D^*)$ where $\sqsubseteq_D^*$ is defined as the minimal binary relation satisfying

1. $p \sqsubseteq_D^* p$
2. $p \sqsubseteq_D^* q, q \sqsubseteq_D^* r$ implies $p \sqsubseteq_D^* r$
3. $p \sqsubseteq_D q$ implies $p \sqsubseteq_D^* q$
4. $p \sqsubseteq_D^* q, r \sqsubseteq_D^* s$ and $\overrightarrow{p} = \overrightarrow{r}$ implies $(p;r) \sqsubseteq_D^* (q;s)$

Theorem 4.1 Composition respects all kinds of simulation.
Proof: Similar to lemma 3.7.

Now let us examine down-simulation constructors.
Theorem 4.2 Down-simulations respect up-simulation and total simulation.
Proof: Let $A, C : D \to E$ be functors and $d : A > C$ a down-simulation. For any object $h(b_1, .., b_k)$ in $\Phi(D)$ define

$$\Phi^+(d)h(b_1, .., b_k) \stackrel{def}{=} h(db_1, .., db_n)$$

Now we are going to prove that $\Phi^+(d) : \Phi(A) > \Phi(C)$.
(1) σ is an endofunctorial constructor h.

$$\begin{array}{lll}
 & \Phi(A)h(p_1, .., p_k); \Phi^+(d)\ h(\overrightarrow{p_1, .., p_k} & \\
= & h(Ap_1, .., Ap_k); \Phi^+(d)h(\vec{p_1}, .., \vec{p_k}) & \{def\ of\ \Phi\} \\
= & h(Ap_1, .., Ap_k); h(d\ \vec{p_1}, .., d\ \vec{p_k}) & \{def\ of\ \Phi^+\} \\
= & h(Ap_1; d\ \vec{p_1}, .., Ap_k; d\ \vec{p_k}) & \{h\ isd\ a\ functor\} \\
\sqsubseteq & h(d\ \vec{p_1}; Cp_1, .., d\ \vec{p_k}; Cp_k) & \{d : A > C\} \\
= & \Phi^+(d)\ h(\overleftarrow{p_1, .., p_k}); \Phi(C)h(p_1, .., p_k) & \{by\ a\ mirror\ argument\}
\end{array}$$

(2) σ is an up-simulation constructor $m : h_1 < h_2$.

$$\begin{array}{lll}
 & \Phi(A)m(b_1, .., b_k); \Phi^+(d)\ m(\overrightarrow{b_1, .., b_k}) & \\
= & m(Ab_1, .., Ab_k); \Phi^+(d)h_2(b_1, .., b_k) & \{def\ of\ \Phi\} \\
= & m(d\ \overleftarrow{p_1}, .., d\ \overleftarrow{p_k}); h_2(db_1, .., db_k) & \{def\ of\ \Phi^+\} \\
\sqsubseteq & h_2(db_1, .., db_k); m(d\ \vec{p_1}, .., d\ \vec{p_k}) & \{m : h_1 < h_2\} \\
= & \Phi^+(d)\ m(\overleftarrow{b_1, .., b_k}); \Phi(C)m(b_1, .., b_k) & \{by\ a\ mirror\ argument\}
\end{array}$$

(3) σ is the composition constructor ;.

$$\begin{array}{lll}
 & \Phi(A)(p_1; p_2); \Phi^+(d)\ (\overrightarrow{p_1; p_2}) & \\
= & (Ap_1; Ap_2); d\ \vec{p_2} & \{def\ of\ \Phi\} \\
\sqsubseteq & Ap_1; d\ \overleftarrow{p_2}; Cp_2 & \{d : A > C\} \\
\sqsubseteq & d\ \vec{p_1}; Cp_1; Cp_2 & \{d : A > C\} \\
= & \Phi^+(d)\ (\overleftarrow{p_1; p_2}); \Phi(C)(p_1; p_2) & \{by\ a\ mirror\ argument\}
\end{array}$$

From theorem 4.1 it follows that $\Phi^+(d)$ is a down-simulation from $\Phi(A)$ to $\Phi(C)$.

If (d, u) is a total simulation from C to A, define

$$\begin{array}{lll}
\Phi^+(d)h(b_1, .., b_k) & \stackrel{def}{=} & h(db_1, ...db_n) \\
\Phi^+(u)h(b_1, .., b_k) & \stackrel{def}{=} & h(ub_1, .., ub_n)
\end{array}$$

Because h is a covariant functor we have

$$\begin{array}{lll}
 & \Phi^+(d)h(b_1, .., b_k); \Phi^+(u)h(b_1, .., b_k) & \\
= & h(db_1, .., db_k); h(ub_1, .., ub_k) & \{def\ of\ \Phi\} \\
= & h(db_1; ub_1, .., db_k; ub_k) & \{h\ is\ a\ functor\} \\
= & h(Ab_1, .., Ab_k) & \{(d, u) : C \to A\} \\
= & \Phi(A)h(b_1, .., b_k) & \{def\ of\ \Phi\}
\end{array}$$

$$
\begin{array}{lll}
 & \Phi^+(u)h(b_1,..,b_k);\Phi^+(d)h(b_1,..,b_k) & \\
= & h(ub_1,..,ub_k);h(db_1,..,db_k) & \{def\ of\ \Phi\} \\
= & h(ub_1;db_1,..,ub_k;db_k) & \{h\ is\ a\ functor\} \\
\sqsubseteq & h(Cb_1,..,Cb_k) & \{(d,u):C \rightarrow A\} \\
= & \Phi(C)h(b_1,..,b_k) & \{def\ of\ \Phi\}
\end{array}
$$

Moreover, the previous argument shows that $\Phi^+(d) : \Phi(A) > \Phi(C)$. By appealing to lemma 4.1 it follows that $(\Phi^+(d), \Phi^+(u))$ is a total simulation.

It is routine to check that Φ^+ preserves horizontal and vertical compositions of down-simulations. Therefore Φ is the thinning of a quasi 2-functor Φ^+.

The following theorem is dual to theorem 4.2.
Theorem 4.3 Up-simulations respect down-simulation and total simulation.

A covariant natural transformation can be regarded as both a down-simulation and an up-simulation. From lemma 4.5 it follows that it respects all kinds of simulation.
Theorem 4.4 Covariant natural transformations respect all kinds of simulation.

Let us now consider a function h which satisfies the distributive law:

$$
\begin{array}{l}
hp : h\ \vec{p} \rightarrow h\ \overleftarrow{p} \\
h(p;q) = hp;hq
\end{array}
$$

Because distribution of h through composition reverses the order of the operands, it is known as a *contravariant functor*.
Theorem 4.6 Contravariant functorial constructors respect total simulation.
Proof: Let $(d, u) : C \rightarrow A$ be a total simulation. Define for any contravariant functor h

$$
\begin{array}{lll}
\Phi^+(d)h(b) & \stackrel{def}{=} & h(ub) \\
\Phi^+(u)h(b) & \stackrel{def}{=} & h(db)
\end{array}
$$

Then one has

$$
\begin{array}{lll}
 & \Phi(A)(hp);\Phi^+(d)\ \overleftarrow{hp} & \\
= & h(Ap);\Phi^+(d)(h\ \overleftarrow{p}) & \{def\ of\ \Phi\} \\
= & h(Ap);h(u\ \overleftarrow{p}) & \{def\ of\ \Phi^+\} \\
= & h(u\ \overleftarrow{p};Ap) & \{h\ is\ contravariant\} \\
\sqsubseteq & h(Cp;u\ \vec{p}) & \{u : C < A\} \\
= & \Phi^+(d)\ \overleftarrow{hp};\Phi(C)(hp) &
\end{array}
$$

$$
\begin{array}{lll}
 & \Phi^+(d)hb;\Phi^+(u)hb & \\
= & h(ub);h(db) & \{def\ of\ \Phi^+\} \\
= & h(db;ub) & \{h\ is\ contravariant\} \\
= & h(Ab) & \{(d,u):C \rightarrow A\} \\
= & \Phi(A)(hb) & \{def\ of\ \Phi\}
\end{array}
$$

$$\Phi^+(u)hb;\Phi^+(d)hb$$

$$
\begin{array}{lll}
= & h(db);h(ub) & \{def\ of\ \Phi^+\} \\
= & h(ub;db) & \{h\ is\ contravariant\} \\
\sqsubseteq & h(Cb) & \{(d,u):C \rightarrow A\} \\
= & \Phi(C)(hb) & \{def\ of\ \Phi\}
\end{array}
$$

From lemma 4.1 it follows that $(\Phi^+(d), \Phi^+(u))$ is a total simulation.

Theorem 4.7 Contravariant natural transformations respect total simulation.
Proof: Similar to theorem 4.6.

Let $f : L \rightarrow M$ and $g : M \rightarrow L$ be covariant functors. We define a *junction* as a weaker form of an adjunction, which does not need to be a bijection. A *right junction* Θ from f to g is a function of three arguments; the first is an identity in L, the second is an identity in M, and the third an arrow in L. The result of Θ is an arrow of M. More precisely, if $q : b \rightarrow gc$ in L then $\Theta bcq : fb \rightarrow c$ in M. Furthermore, Θ satisfies

1. $\Theta\ \overleftarrow{p}\,\overrightarrow{q}\ (p;q) = fp;\Theta\ \overrightarrow{p}\,\overrightarrow{q}\ q$ for all p, q in L of appropriate type.

2. $\Theta\ \overleftarrow{q}\,\overrightarrow{r}\ (q;gr) = (\Theta\ \overleftarrow{q}\,\overleftarrow{r}\ q);r$ for all q in L and r in M of appropriate type.

Having defined a preorder in the homsets of the categories L and M, the above equations can then be replaced hy inequations. A *right-down-junction* Θ_{down} from f to g is a function possessing the following properties:

1. $fp;\Theta_{down}\ \overrightarrow{p}\,\overrightarrow{q}\ q \sqsubseteq \Theta_{down}\ \overleftarrow{p}\,\overrightarrow{q}\ (p;q)$

2. $\Theta_{down}\ \overleftarrow{q}\,\overrightarrow{r}\ (q;gr) \sqsubseteq (\Theta_{down}\ \overleftarrow{q}\,\overleftarrow{r}\ q);r$

A *right-up-junction* Θ_{up} from f to g satisfies

1. $\Theta_{up}\ \overleftarrow{p}\,\overrightarrow{q}\ (p;q) \sqsubseteq fp;\Theta_{up}\ \overrightarrow{p}\,\overrightarrow{q}\ q$

2. $(\Theta_{up}\ \overleftarrow{q}\,\overleftarrow{r}\ q);r \sqsubseteq \Theta_{up}\ \overleftarrow{q}\,\overrightarrow{r}\ (q;gr)$

It is clear that a right junction Θ is hoth a right-up-junction and a right-down-junction.

The concept of a *left junction* Ψ from g to f is dual to that of a right junction. For any arrow $q : fb \rightarrow c$ in the category M, Ψbcq is an arrow in L with b as its source, and with gc as its target. Ψ satisfies

1. $\Psi\ \overleftarrow{q}\,\overleftarrow{r}\ (q;r) = (\Psi\ \overleftarrow{q}\,\overleftarrow{r}\ q);gr.$

2. $\Psi\ \overleftarrow{p}\,\overrightarrow{q}\ (fp;q) = p;(\Psi\ \overrightarrow{p}\,\overrightarrow{q}\ q)$

In analogy with what we did for right junction, we can definea so-called *left-down-junction* and *left-up-junction*. the former satisfies

1. $p;(\Psi_{down}\ \overrightarrow{p}\,\overrightarrow{q}\ q) \sqsubseteq \Psi_{down}\ \overleftarrow{p}\,\overrightarrow{q}\ (fp;q)$

2. $\Psi_{down}\ \overleftarrow{q}\,\overrightarrow{r}\ (q;r) \sqsubseteq (\Psi_{down}\ \overleftarrow{q}\,\overleftarrow{r}\ q);gr$

The latter possesses the following properties

1. $\Psi_{up}\ \overleftarrow{p}\,\overrightarrow{q}\ (fp;q) \sqsubseteq p;(\Psi_{up}\ \overrightarrow{p}\,\overrightarrow{q}\ q)$

2. $(\Psi_{up}\ \overleftarrow{q}\,\overleftarrow{r}\ q);gr \sqsubseteq \Psi_{up}\ \overleftarrow{q}\,\overrightarrow{r}\ (q;r)$

The introduction of junctional constructors into a programming language maintains validity of simulations as shown below.

Theorem 4.8 Down-junctional constructors respect up-simulation and total simulation.

Proof: Let u be an up-simulation from C to A. Define for any right-down-junction Θ_{down} from f to g

$$\Phi^+(u)\ \Theta_{down}\overrightarrow{bcp} \stackrel{def}{=} uc$$
$$\Phi^+(u)\ \Theta_{down}\overleftarrow{bcp} \stackrel{def}{=} fub$$

Then one has

$$\begin{aligned}
& \Phi^+(u)\ \Theta_{down}\overleftarrow{bcp}; \Phi(A)(\Theta_{down}bcp) \\
= \ & f(ub); \Theta_{down}(AbAcAp) && \{def\ of\ \Phi^+\} \\
\sqsubseteq \ & \Theta_{down}(CbAc(ub; Ap)) && \{the\ property\ (1)\ of\ right-down-junction\} \\
\sqsubseteq \ & \Theta_{down}(CbAc(Cp; g(uc))) && \{u : C < A\} \\
\sqsubseteq \ & \Theta_{down}(CbCcCp); uc && \{the\ property\ (2)\ of\ right-down-junction\} \\
= \ & \Phi(C)(\Theta_{down}bcp); \Phi^+(u)\ \Theta_{down}\overrightarrow{bcp} && \{def\ of\ \Phi^+\}
\end{aligned}$$

as required.

Left-down-junction can be treated in a similar way.

Theorem 4.9 Up-junctional constructors respect down-simulation and total simulation.

Proof: Similar to that of theorem 4.8.

Theorem 4.10 Junctional constructors respect all kinds of simulation.

Proof: Direct from lemma 4.5, theorem 4.9 and 4.10.

The concept of junction can extend to the contravariant functors. Suppose that f is a covariant bifunctor, and g is contravariant in its first argument and covariant in the second argument. A contravariant junction Υ from f to g is a function satisfying the following properties

1. If $q : f(b, c) \rightarrow a$ in M then $\Upsilon bcaq : b \rightarrow g(c, a)$ in L.
2. $\Upsilon\ \overleftarrow{p}\ \overleftarrow{q}\ \overrightarrow{r}\ (f(p, q); s; r) = p; \Upsilon\ \overrightarrow{p}\ \overrightarrow{q}\ \overleftarrow{r}\ s; g(q, r)$

Theorem 4.11 Contravariant junctional constructors respect total simulation.

Proof: Define for any contravariant junction Υ from f to g

$$\Phi^+(u)\ \Upsilon\overrightarrow{bcap} \stackrel{def}{=} g(dc, ua)$$
$$\Phi^+(u)\ \Upsilon\overleftarrow{bcap} \stackrel{def}{=} ub$$
$$\Phi^+(d)\ \Upsilon\overrightarrow{bcap} \stackrel{def}{=} g(uc, da)$$
$$\Phi^+(d)\ \Upsilon\overleftarrow{bcap} \stackrel{def}{=} db$$

The conclusion can be established by the techniques similar to those used in the previous theorem.

5 Language Constructors

In this section we show that many constructors in a range of programming languages have familiar categorical interpretations. For simplicity, we will suppress mention of types (objects) whenever possible.

5.1 Composition

In all programming languages of interest, there exists a composition operator (denoted here as $p;q$, elsewhere $p * q$). Execution of such a composite program usually (but not always) involves execution of both of its components. In a procedural programming language like **PASCAL**, we interpret this notation as *sequential execution*: q does not start until p has successfully terminated. In a functional language it denotes functional composition. This operator is associative

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

It has both a left and a right unit. In Dijkstra's language [1], the unit is the command **skip**

$$\mathbf{skip};p \;=\; p;\mathbf{skip} \;=\; p$$

In a typed language, the composition of programs is undefined when the type of the result of the first component differs from that expected by the second component. This can be treated in category theory by associating source and target types with each program. $(p;q)$ is then well-defined iff $\vec{p} = \overleftarrow{q}$.

In the rest of this section we assume without explicit mention that all type constraints have been observed.

A *zero* of composition (if it exists) is denoted by $\emptyset$. It is the program that fails to terminate. The defining property of the zero program is

$$p;\emptyset \;=\; \emptyset \;=\; \emptyset;q$$

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

In Dijkstra's language, this role of zero program is played by the program **abort** which is the *bottom element* in its homset. It may fail to terminate; or being non-deterministic it may do even worse: it may terminate with the wrong result, or even the right one (sometimes, just to mislead you). To specify the execution of q after termination of **abort** cannot redeem the situation, because **abort** cannot be relied on to terminate. To specify execution of p before abortion is equally ineffective, because the non-termination will make any result of executing p inaccessible and unusable. In other words, composition in Dijkstra's language is *strict* in the sense that it gives bottom if either of its arguments is bottom. The above defining equation states that zero program is a natural transformation.

A language like **CSP** [4] 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 the above equation. 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, the defining property of the aborting command must replaced by

$$\emptyset ; q \;=\; \emptyset \;\sqsubseteq\; p ; \emptyset$$

which states that $\emptyset$ is an up-simulation.

In a lazy functional programming language like Miranda [9], the call of a function will not evaluate an argument unless the value of the argument is actually needed during execution of the body of the function. As a result, it may terminate even when applied to a non-terminating argument. However, the wholly undefined function always fails. On the principle the failure is worse than any kind of success, the property of zero program has to bereplaced by

$$p ; \emptyset \;=\; \emptyset \;\sqsubseteq\; \emptyset ; q$$

i.e., zero programs become a down-simulation in this case.

We use $p \sqcap q$ to denote the best common approximation in the $\sqsubseteq$ ordering of both p and q, if it exists. It can be defined by the single law

$$r \sqsubseteq (p \sqcap q) \;\; \textit{iff} \;\; r \sqsubseteq p \;\textit{and}\; r \sqsubseteq q$$

We are going to explore the way in which composition interacts with the $\sqcap$ operator. From the defining property of $\sqcap$ and the monotonicity of composition we can derive the following weak distributive law

$$r ; (p \sqcap q) ; s \;\sqsubseteq\; (r ; p ; s) \sqcap (r ; q ; s)$$

In Dijkstra's language (and other truly non-deterministic language like CSP), $\sqcap$ denotes non-determinism; and the law can be strengthened to an equation

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

This law states that it makes no difference whether the selection between p and q is made before execution of the first operand of a composition (e.g., at compiler time), or whether it is made (at run time) after execution of the first operand. In other words, the $\sqcap$ is a junction.

However, in a functional or deterministic language it is better to postpone the application of $\sqcap$ as long as possible, because it somehow worsens its argument. The above strengthening is not valid, instead we have

$$\begin{aligned} (p \sqcap q) ; s \;&\sqsubseteq\; (p ; s) \sqcap (q ; s) \\ r ; (p \sqcap q) \;&=\; (r ; p) \sqcap (r ; q) \end{aligned}$$

In this case, the $\sqcap$ operator is a quasi-junction.

5.2 Disjoint Union

The coproduct (disjoint union) constructor 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 $(\overrightarrow{p} + \overrightarrow{q})$ it first tests which variant it comes from. If it is the first variant, then p is applied, obtaining a result of type $\overrightarrow{p}$, which is then injected into the first variant of $(\overrightarrow{p} + \overrightarrow{q})$. The treatment of the second case is similar. Thus

$$\begin{aligned} (\overrightarrow{p + q}) &= \overrightarrow{p} + \overrightarrow{q} \\ (\overleftarrow{p + q}) &= \overleftarrow{p} + \overleftarrow{q} \end{aligned}$$

Furthermore, it is easy to see that the above description of the case discrimination satisfies the other defining property of a bifunctor

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

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 N to the disjoint union $\mathrm{N} + \mathrm{N}$. When applied to an even number , say $2n$, its result $(0, 2n)$ is the same number tagged as in the first alternative of the discriminated union; whereas an odd number $2n + 1$ 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

$$even\,;(halve + add)$$

But it still remains to map the result of this conditional from the discriminated union $\mathrm{N} + \mathrm{N}$ back to the single natural number type N. For this we need for each type b, a *merge* operator symbolised by ∇b, 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

$$\mathbf{if}\ even(x)\ \mathbf{then}\ x := x/2\ \mathbf{else}\ x := x + 1\ \mathbf{fi}$$

the conditional described above should be completed as follows

$$even\,;(halve + add)\,;\nabla\mathrm{N}$$

If p maps b to c, p may be applied after the merging operation ∇b, or it may be applied to both alternatives before the merging operator ∇c; the final result of each of these applications will be the same. Thus merging operator satisfies

$$(p + p);\ \nabla\,\overrightarrow{p} = \nabla\,\overleftarrow{p}\,;p$$

The above algebraic law states that ∇ is a natural transformation between the identity functor and the functor that maps p to $(p + p)$.

In a programming language, there are two extreme conditions for each pair of types b and c, $true_{b,c} : b \to (b + c)$ and $false_{b,c} : c \to (b + c)$:

- $true_{b,c}$ which tags its argument as the first alternative of type $b + c$.
- $false_{b,c}$ which tags its argument as the second alternatives of type $b + c$

These are called insertion functions. Thus if $(p + q)$ is executed after $true_{\overline{p},\overline{q}}$, the first alternative p is invariably selected; so the effect is the same as if p had been applied beforehand

$$true_{\overline{p},\overline{q}}\,;(p + q) \;=\; p\,; true_{\overline{p},\overline{q}}$$

Similarly

$$false_{\overline{p},\overline{q}}\,;(p + q) \;=\; q\,; false_{\overline{p},\overline{q}}$$

Thus both condition *true* and condition *false* are natural transformations. Furthermore they satisfy

$$\begin{aligned} true_{b,b}\,;\nabla b &= id_b \\ false_{b,b}\,;\nabla b &= id_b \\ (true_{b,c} + false_{b,c})\,;\nabla(b + c) &= id_{b+c} \end{aligned}$$

where id_b stands for the identity fnnction on the type b.

However, in a non-strict programming language the discriminated union of types b and c is not simply the disjoint sum of b and c as described before, but is defined by

$$b + c \stackrel{\text{def}}{=} \{\bot\} \cup \{(x, 0) \mid x \in b\} \cup \{(y, 1) \mid y \in c\}$$

where a new element $\bot$, represents the bottom element of the union type. The program $p + q$ will map $(x, 0)$ where $x \in \overline{p}$ to $(px, 0)$, and $(y, 1)$ where $y \in \overline{q}$ to $(qy, 1)$, and the bottom element $\bot$ to $\bot$. The merging operator ∇ will be defined by

$$\begin{aligned} \nabla b : (b + b) &\rightarrow b \\ (x, 0) &\mapsto x \\ (y, 1) &\mapsto y \\ \bot &\mapsto \bot \end{aligned}$$

In this language, $+$ is a *quasi-coproduct* [5], in a sense defined up to equivalence by the laws previously given for coproduct except that the merging operator is a downward simulation, and governed by

$$(p + p)\,;\nabla\,\overline{p} \sqsubseteq \nabla\,\overline{p}\,;p$$

This is because the program $(p + p)\,;\nabla\,\overline{p}$ will map $\bot$ to $\bot$, but $\nabla\,\overline{p}\,;p$ will not so when the program is non-strict.

5.3 Product

A similar treatment can be given to the product bifunctor $p \times q$, where programs p and q are assumed to be run in parallel without interference. The associated natural transformations are the *projections* $\pi_{b,c} : (b \times c) \rightarrow b$ and $\mu_{b,c} ; (b \times c) \rightarrow c$, and the *duplicating* operator $\Delta b : b \rightarrow (b \times b)$, which maps x of type b to the pair (x, x). In a category of total functions, they satisfy

$$\begin{aligned} (p \times q) ; \pi_{\overleftarrow{p},\overleftarrow{q}} &= \pi_{\overleftarrow{p},\overleftarrow{q}} ; p \\ (p \times q) ; \mu_{\overleftarrow{p},\overleftarrow{q}} &= \mu_{\overleftarrow{p},\overleftarrow{q}} ; q \\ \Delta \overleftarrow{p} ; (p \times p) &= p ; \Delta \overrightarrow{p} \end{aligned}$$

Let p and q be programs with $\overleftarrow{p} = \overleftarrow{q}$, we define their *product* $< p, q >$ to be a program which makes a second copy of the current argument, and execute p on one of the two copies and q on the other one, and delivers the two results as a pair. In a functional programming language with lists as a data structure, this can be defined:

$$< p, q > \stackrel{\text{def}}{=} \lambda x . cons(px, qx)$$

In a categorical setting it can be formulated by

$$< p, q > \stackrel{\text{def}}{=} \Delta \overleftarrow{p} ; (p \times q)$$

From the defining properties of Δ and bifunctor $\times$ it follows that

$$< p;q;r,\ p;s;t > = p ; < q, s > ; (r \times t)$$

This states that the product function is actually a left junction from the duplicating functor, that maps p to a pair (p, p), to the bifunctor $\times$.

But in many language the above equations do not hold. Suppose that the calculation on q fails to terminate. Then the execution of $(p \times q) ; \pi_{\overleftarrow{p},\overleftarrow{q}}$ in a strict language like **LISP** will also fail to terminate. The program $\pi_{\overleftarrow{p},\overleftarrow{q}} ; p$ does not involve an operation on the discarded alternative q, and will therefore terminate in cases $(p \times q) ; \pi_{\overleftarrow{p},\overleftarrow{q}}$ will not. This can be expressed mathmatically by inequations stating that the projections π and μ are downward simulations from the product bifunctor to the bifunctor that selects one of its operands,

$$\begin{aligned} (p \times q) ; \pi_{\overleftarrow{p},\overleftarrow{q}} &\sqsubseteq \pi_{\overleftarrow{p},\overleftarrow{q}} ; p \\ (p \times q) ; \mu_{\overleftarrow{p},\overleftarrow{q}} &\sqsubseteq \mu_{\overleftarrow{p},\overleftarrow{q}} ; q \end{aligned}$$

The strong equations, of course, remain true for a lazy functional language, in which no result is computed until it is known to be needed.

In a programming language which permits non-determinism, the duplicating operator does not satisfy the equation $\Delta \overleftarrow{p} ; (p \times p) = p ; \Delta \overrightarrow{p}$. 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 on 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 [3]

$$\triangle \overline{p} \,;\, (p \times p) \sqsubseteq p \,;\, \triangle \overline{p}$$

Consequently one has

$$\begin{array}{lcl} < p;q,p;r > & \sqsubseteq & p \,;\, < q, r > \\ < q;r,s;t > & = & < q, s > \,;\, (r \times t) \end{array}$$

5.4 Higher Order Functions

As useful example of a bifunctor of mixed variance is the exponental 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 the type consistency equires that f must be in $(\overrightarrow{p} \Rightarrow \overleftarrow{q})$ and the result will he in $(\overleftarrow{p} \Rightarrow \overrightarrow{q})$. So

$$(p \Rightarrow q) \,:\, (\overrightarrow{p} \Rightarrow \overleftarrow{q}) \longrightarrow (\overleftarrow{p} \Rightarrow \overrightarrow{q})$$

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, the bifunctor $\Rightarrow$ is contravariant in its first operand, covariant in its second.

Consider a function $f \,:\, (b \times c) \rightarrow a$, which takes a pair of arguments. The curried version of f is the same as f, except that it takes its arguments one at time. Thus $(curry\, f) \,:\, b \rightarrow (c \Rightarrow a)$ 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

$$((curry\, f)\, x)\, y) \;=\; f(x, y)$$

In category theory use of variable is forbidden; furthermore, the operator needs to be subscripted by the types of its operands and is characterized by the following laws

$curry_{b,c,d}(f) \,:\, b \rightarrow (c \Rightarrow a)$ for $f \,:\, (b \times c) \rightarrow a$
$curry_{\overrightarrow{p},\overleftarrow{q},\overleftarrow{r}}((p \times q); f; r) \;=\; p; curry_{\overrightarrow{p},\overrightarrow{q},\overleftarrow{r}}(f); (q \Rightarrow r)$

This states that *curry* is a contravariant junction from the covariant bifunctor $\times$ to the mix-variant bifunctor $\Rightarrow$.

The currying operator has an inverse called *uncurrying*. Its defining properties are

$$uncurry_{b,c,d}(f) : (b \times c) \rightarrow a \text{ for } f : b \rightarrow (c \Rightarrow a)$$
$$uncurry_{\overleftarrow{p},\overrightarrow{q},\overleftarrow{r}}(p; f; (q \Rightarrow r)) \;=\; (p \times q); curry_{\overleftarrow{p},\overrightarrow{q},\overleftarrow{r}}(f); r$$

5.5 Recursive Programs

Let Ψ be a continuous constructor satisfying for any program p

$$\Psi(p) : \overleftarrow{p} \rightarrow \overrightarrow{p}$$

The recursive program $\mu x_{b,c}.\, \Psi(x_{b,c})$ is defined in e.g., [8] as the least upper bound of the ascending chain

$$\emptyset_{b,c} \sqsubseteq \Psi(\emptyset_{b,c}) \sqsubseteq \Psi^2(\emptyset_{b,c}) \sqsubseteq \ldots.$$

where $\emptyset_{b,c}$ denotes the worst program with the source type b and the target type c.

From the property of the least upper bound operator $\sqcup_n$ we can derive for any ascending chain $\{p_n\}$

$$\sqcup_n (p_n ; q) \;\sqsubseteq\; \sqcup_n(p_n) ; q$$

This law states that the least upper bound operator is a quasi-junction.

In Dijkstra's language the loop program **do** $b \rightarrow p$ **od** is defined as the least fixed point of the recursive equation

$$x \;=\; \textbf{if } b \textbf{ then } p; x \textbf{ else skip fi}$$

6 Conclusion

This paper has looked at a categorical approach to the theory of data refinement. The goal is to explore the sufficient conditions for the validity of data refinement by various simulations, and to relate them to familiar categorical concepts.

Data refinement is known to be an important method for designing computer programs as well as implementation of computer programming languages. It is therefore important to have simple proof methods to proof its correctness, and to know what methods are valid for various kinds of language in use. For example, in a first-order programming language (without procedures or functions as parameters) the simple proofs work for natural transformations, but in

a higher-order language they work only for the more restricted class of total simulations. We have investigated the relationship between the validity of data refinement and the properties of language constructors. After getting a clear view of many useful feature of programming languages, we know the reason why those constructors are to be recommended and why some other are not. The result of this paper provides an important criterion for design of a new programming language, that it should maintain the validity of some clearly defined technique of data refinement.

Acknowledgement

To Wim Hesselink, Joseph Goguen, Martín Hyland, Peter Freyd and Samson Abramski for assistance, encouragement and advice of various kinds. Also to the Admiral B.R. Inman Centennial Chair in Computing Theory at the University of Texas at Austin for support during the studies which led to this paper. The research was also supported in part by the Science and Engineering Research Council of Great Britain.

References

[1] E.W. Dijkstra, *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs. NJ, (1976).

[2] J.W. Gray, *Formal Category Theory: Adjointness for 2-categories.* LNM 391, Springer-Verlag, (1974).

[3] M. Hennessy, *The semantics of call-by-value and call-by-name in a non-deterministic environment.* SIAM J. Comp. (1980), 67–85.

[4] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, (1985).

[5] C.A.R. Hoare and He Jifeug, *Two-categorical Semantics for Programming Languages.* in preparation.

[6] J. Lambek and P.J. Scott, *Introduction to higher order categorical logic* Cambridge University Press, (1985).

[7] Sanders Mac Lane, *Categories for the working mathematicians.* Springer-Verlag, New York Inc. (1971).

[8] D.S. Scott, *The lattice of flow diagrams.* Symposium on Semantics of Algorithmic Languages, LNM 118, E. Engeler (ed.), (1971) 311–366.

[9] D.A. Turner, *Miranda, a non-strict functional language with polymorphic types.* LNCS 201, Springer-Verlag, (1985) 1–16.

Prespecification and Data Refinement

He Jifeng, C.A.R. Hoare

September 4, 1990

Contents

1 Introduction

A data type is generally defined, in a manner similar to an algebra, as a set of values together with a family of operations on these values. The operations are indexed by procedure names, usually with parameters for conveying values and results between the data type and the using program. It is only by employing these procedures that the using program can update and interrogate the value of a variable of the given type.

One data type (call it concrete) is said to refine a data type with the same index set (call it abstract) if in all circumstances and all purposes the concrete type can be validly used in place of the abstract one. The practical benefit of this arises when the abstract data type can be specified, understood and used in an applications program but can not directly or efficiently represented on a computer; whereas the concrete type is some efficient representation of the abstract one involving perhaps a complicated collection of bitmaps, pagetables and fileblocks, which can be economically stored and updated.

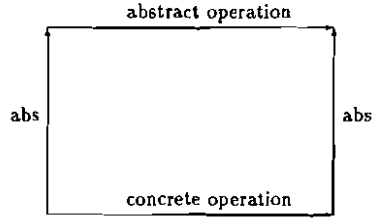
Data refinement technology plays a crucial role in designing programs. It enables us to write programs based on abstract data type easily and elegantly, and to derive efficient programs based on sophiscated concrete data types effiectively. Much research in this area has produced various kinds of refinement rules [2,3,4,7,8]. An early suggestion for a method of data refinement was given in [4]. The method was based upon

(1) an invariant predicate which must be proved true after initialisation and after every operation on structure, assuming that it was true beforehand.

(2) an abstract function which maps the current value of the concrete data type onto the abstract value which it stands for. The abstract function must be proved to commute with all the operations of the data type in the following sense:

> To apply the abstract function after a concrete operation gives the same result as applying the abstract function before the corresponding abstract operation

This is sometimes expressed as a commuting diagram in which abs is the abstract function



This method was adopted and developed in the VDM techniqne of data refinement [7]. In VDM, certain additional properties of a data type are considered desirable.

(1) The abstract data type should be fully abstract. This means that any two distinct values of the abstract data type can be distinguished by some sequence of operations on the data.

(2) The concrete data type should be adequate to represent every value of the abstract data type, that is, the abstraction function should be surjective.

In this paper we attempt simultaneously to generalise and simplify the notion of data refinement in the following ways

(1) Both the abstract and the concrete operations may be nondeterminstic. We will use relations to represent the commands over the data type.

(2) There is no need for the concepts of full abstraction or adequacy.

(3) The relationship between the concrete and abstract data types does not have to be functional; the invariant and the abstraction relation will be combined into a single relation called a simulation. A simulation may be either upward (concrete-to-abstract) or downward (abstract to concrete). The two kinds of simulations are sufficient for data refinement and together they are necessary. This is a new result for nondeterminstic programs.

(4) The simulations rules will enable us to calculate the weakest specification of each operation on the concrete type from the operation on the data type and the simulations.

The following relational notation will be used in the later discussion. Let S be the set of states of a system. We shall describe an operation on the system by using a binary relation on S: the state of the system before the operation is denoted s and the state after denoted s'. Important notations include

$$\begin{aligned} U_S &= S \times S \\ I_S &= \{(s,s') : S \times S \mid s = s'\} \\ \breve{R} &= \{(s',s) : S \times S \mid (s,s') \in R\} \\ \bar{R} &= \{(s,s') : S \times S \mid (s,s') \notin R\} \end{aligned}$$

$R \cup T$, $R \cap T$, $R \subseteq T$ *and* $R;T$ denotes the union, intersection, containment and forward relational composition of R and T respectively.

Our definitions and proofs will be considerably simplified by confining attention to total relation, in which case $R \subseteq T$ means simply R is at least as deterministic as T. Tbe justification for this simplification can be found, for example, in [5].

We find it convenient to have notion for the weakest amongest both the first and the second of a pair of relations whose composition meets some specification. We define the weakest postspecification of relations (see [5]) as

$$R/T = \overline{(\breve{T};\overline{R})}$$

The definition is difficult to explain and to use; for most purposes it is sufficient to recall that weakest postspecification is an approximate left inverse of composition in the following sense [5]

$$T;X \subseteq R \;\equiv\; X \subseteq R/T$$

Analogously the weakest prespecification is defined [5] as

$$T\backslash R = \overline{(\overline{R}; \breve{T})}$$

which is characterized by the law

$$X;T \subseteq R \equiv X \subseteq T\backslash R$$

2 Data Types

A data type **A** is defined in a fairly conventional manner to be a quadruple

$$\mathbf{A} = (AVAL, AI, A, AF)$$

where $AVAL$ is the space of values of the type. AI is an initialisation operation, which is a relation from some global data space to $AVAL$; and AF is a finalisation operation which is a relation from $AVAL$ back to the same global data space. $A = \{aop_i \mid i \in I\}$ is an indexed set of relations over $AVAL$; total relations in A represent commands that npdate or interrogate the data, and partial relations represent guarded commands, guarded by a condition that is just true on the domain of relation. Nontermination must therefore be represented by some fictitious value $\perp$ appended to the set $AVAL$ and mapped to everything by each command.

A data type A is said to be canonical if all the operations aop_i and AI are functions (total or partial). Programs will be written in an anologue of guarded commands [1]. This is restrictive enough to be implemented efficiently yet powerful enough to include nondeterminism and recursion. The set of programs over the data type **A** is defined to be the smallest set $\mathcal{D}(\mathbf{A})$ containing

(1) the universal relation U and the identity relation I over the related data space
(2) all operations aop_i in **A**
(3) $P; Q$ and $P \cup Q$ for any P and Q in $\mathcal{D}(\mathbf{A})$
(4)$\bigcap_n P_n$ where the P_n form a descending chain of total elements of $\mathcal{D}(\mathbf{A})$; that is, for all n, $P_{n+1} \subseteq P_n$

A complete program over the data type **A** is one which begins with initialisation and ends with finalisation. The space of all complete program over **A** is thus defined to be

$$\mathcal{PROG}(\mathbf{A}) = \{AI; P; AF \mid P \in \mathcal{D}(\mathbf{A})\}$$

A non-empty subset of a data type is called finitary if it is either finite or the whole type. A relation on the type is called finitary if the image of each element is finitary. In order to ensure proper convergence under clause (4), we insist that all relations in **A** be finitary; this property is preserved by all programs in $\mathcal{D}(\mathbf{A})$ and $\mathcal{PROG}(\mathbf{A})$

This paper is concerned with various forms of correspondence hetween one data type and another. We consider abstract and concrete data type respectively

$$\mathbf{A} = (AVAL, AI, A, AF)$$

$$\mathbf{C} = (CVAL, CI, C, CF)$$

and we shall assume that these two types are conformal in the sense that

(1) their global data spaces coincide

(2) the indexing sets of A and C coincide.

If $P(\mathrm{A})$ is in $\mathcal{D}(\mathbf{A})$, we write $P(\mathbf{C})$ for that member of $\mathcal{D}(\mathbf{C})$ which is constructed from the corresponding indexd set C in the same way that $P(\mathbf{A})$ was constructed from A. Similarly for any complete program P_A in $\mathcal{PROG}(\mathbf{A})$ we can construct the corresponding complete program P_C in $\mathcal{PROG}(\mathbf{C})$. We shall use the subset ordering on indexed sets, with the obvious meaning,

$$\mathbf{C} \subseteq \mathbf{A} = CI \subseteq AI \wedge CF \subseteq AF \wedge \forall \mathrm{i} \in \mathrm{I}.\ cop_i \subseteq aop_i$$

3 Refinement

Definition. A data type **C** refines a data type **A** if replacement of **A** by **C** in any complete program only reduces that program,that is,

$$CI; P(\mathbf{C}); CF \subseteq AI; P(\mathbf{A}); AF$$

for all $P(\mathbf{A}) \in \mathcal{D}(\mathbf{A})$

Two types that refines each other are said to be equivalent.

Relational containment is used here as a correctness-preserving transformation whose only effect is a possible reduction of nondeterminism. The insistence that all commands are total and the use of a data value $\perp$ to represent nontermination means that total correctness is preserved.

Theorem 1. If $\mathbf{C} \subseteq \mathbf{A}$ then **C** refines **A**
Proof: All operators of the language used in constructing complete programs over **C** and **A** are monotonic in all their arguments. □

Theorem 2. Refinement is transitive: if **C** refines **B** and **B** refines **A**, then **C** refines **A**.
Proof: The proof follows by transitivity of relational containment. □

Refinement is a powerful tool in the design and development of programs, since it permits an abstract algorithm to be designed over some simple abstract type **A**, which is then validly replaced by some complex but efficiently implemented type **C**. However, the definition of refinement gives no indication of how to develop the concrete type: it is something which can be verified, with difficulty, when both **A** and **C** are known. We start by giving two simple proof obligations [3], which can he readily checked and which prove to he sufficient for refinement.

Definition. A downward simnlation is a relation R from $AVAL$ to $CVAL$ satisfying

$$\begin{aligned} CI &\subseteq AI; R \\ R; CF &\subseteq AF \\ R; cop_i &\subseteq aop_i; R \qquad \textit{for each index } i \in \mathbf{I} \end{aligned}$$

Here we insist that R be strict, that is ,

$$\{\perp\} \times CVAL \subseteq R$$

In terms of weakest specification, the inclusions in the above definition become

$$\begin{aligned} CI &\subseteq AI; R \\ CF &\subseteq AF/R \\ cop_i &\subseteq (aop_i; R)/R \qquad \textit{for each index } i \in \mathbf{I} \end{aligned}$$

which provide methods for calculating the specification of **C** from the abstract type **A** and the downward simulations using relational algebra.

Our next concern is with the correctness of the definition of downward simulation for proving refinement.

Theorem 3. If there is a downward simulation R from **A** to **C**, then **C** refines **A**.
Proof. A typical complete program over **C** has the form

$$\begin{array}{rcll} CI; P(\mathbf{C}); CF & \subseteq & (AI; R); P(\mathbf{C}); CF & \textit{the monotonicity of ;} \\ & \subseteq & AI; P(\mathbf{A}); R; CF & \textit{lemma 1 in appendix} \\ & \subseteq & AI; P(\mathbf{A}); AF & \textit{the monotonicity of ;} \end{array}$$

which is a complete program over **A** □

Theorem 4. If R is a downward simulation from **A** to **B**, and T a downward simulation from **B** to **C**, then $R;T$ is a downward simulation from **A** to **C**.
Proof: For each index $i \in I$ we have

$$\begin{array}{rcll} (R;T); cop_i & \subseteq & R; bop_i; T & \\ & & & T \textit{ is a downward simulation from } \mathbf{B} \textit{ to } \mathbf{C} \\ & \subseteq & aop_i; (R;T) & \\ & & & R \textit{ is a downward simulation from } \mathbf{A} \textit{ to } \mathbf{B} \end{array}$$

Other parts can be proved similarly. □

Definition. An upward simulation is a relation L from $CVAL$ to $AVAL$ satisfying

$$\begin{array}{rcll} CI; L & \subseteq & AI & \\ CF & \subseteq & L; AF & \\ cop_i; L & \subseteq & L; aop_i & \textit{for each index } i \in \mathbf{I} \end{array}$$

We insist that L be strict and finitary. The inclusions in the definition are equivalent to

$$\begin{array}{rcll} CI & \subseteq & L \backslash AI & \\ CF & \subseteq & L; AF & \\ cop_i & \subseteq & L \backslash (L; aop_i) & \textit{for each index } i \in \mathbf{I} \end{array}$$

Similarly we can show

Theorem 5. If there is an upward simulation from **C** to **A**, then **C** refines **A**.

Theorem 6. If L is an upward simulation from **C** to **B**, and N is an upward simulation from **B** to **A**, then $L;N$ is an upward simulation from **C** to **A**.

4 Completeness

This section is devoted to the study of the converse property of soundness, namely completeness. The question being asked is therefore: given a refinement **C** of **A**, does there exist a (downward or upward) simulation between **A** and **C**.

The conclusion is : when the data type **A** is canonical, there does exist a downward simulation between **A** and its refinement. Therefore for the canouical data type, downward simulation is both sufficient and necessary for refinement. In general, if the data type **A** is refined by **C**, then there is a data type **CA** such that there are an upward simulation from **CA** to **A** and a downward simulation from **CA** to **C**. This means that downward simulation and upward simulation together are necessary for data refinement.

First we wish to prove that refinement **A** by **C** implies the existence of a downward simulation from **A** to **C** if the data type **A** is canonical.

Theorem 7. When **A** is canonical, downward simulation alone is necessary for refinement.
Proof. Define that for each $P(\mathbf{A})$ in $\mathcal{D}(\mathbf{A})$

$$R(P) = (P(\mathbf{C}); CF) \backslash (P(\mathbf{A}); AF)$$

and let $R = \bigcap_{P \in \mathcal{D}} R(P)$

We shall show that R is a downward simulation from **A** to **C**

(1). For all $P(\mathbf{A})$ in $\mathcal{D}(\mathbf{A})$ we have

$$(CI; P(\mathbf{C}); CF) \subseteq (AI; P(\mathbf{A}); AF)$$

by the assumption. It leads to

$$\begin{array}{lll} CI & \subseteq & (P(\mathbf{C}); CF) \backslash (AI; P(\mathbf{A}); AF) \qquad \textit{def of } \backslash \\ & \subseteq & AI; ((P(\mathbf{C}); CF) \backslash (P(\mathbf{A}); AF)) \qquad \textit{lemma 3 in appendix} \\ & = & AI; R(P) \qquad \textit{def of } R(P) \end{array}$$

Which implies that

$$\begin{array}{lll} CI & \subseteq & \bigcap_{P \in \mathcal{D}} (AI; R(P)) \qquad \textit{set theory} \\ & = & AI; \bigcap_{P \in \mathcal{D}} R(P) \qquad \textit{lemma 5,6 in appendix} \\ & = & AI; R \qquad \textit{def of } R \end{array}$$

(2) Since the identity relation I is a program in $\mathcal{D}(\mathbf{A})$, we conclude

$$\begin{array}{lll} R & \subseteq & R(I) \qquad \textit{set theory and def of } R \\ & = & (I; CF) \backslash (I; AF) \qquad \textit{def of } R(P) \\ & = & CF \backslash AF \qquad I \textit{ is the unit of;} \end{array}$$

which leads to

$$R; CF \subseteq AF \qquad \textit{def of } \backslash$$

(3). From lemma 7 and lemma 8 in appendix it follows that for each index $i \in I$

$$R; cop_i \subseteq aop_i ; R$$

This completes the proof. □

In what follows we will explore a technique by which from any data type **A**, a canonical data type **CA** can be derived such that there exists an upward simulation from **CA** to **A** satisfying for all P in $\mathcal{D}$

$$P_{\mathbf{A}} = P_{\mathbf{CA}}$$

Definition. For any subset B of S and any relation P on S, we define $B\rceil P$ as the image under P of these states in B, i.e.,

$$B\rceil P = \{\tau \mid \exists \rho \in B.\ \rho P \tau\}$$

Now we introduce a relation L from $\mathbf{F}AVAL$ to $AVAL$, where $\mathbf{F}AVAL$ is the family of all finitary subsets of $AVAL$. L is defined by

$$\{B\}\rceil L = B$$

for all finitary subsets B of $AVAL$.

Having defined the relation L we proceed to construct a data type **CA** from the data type **A** and relation L. Here we define

$$CAVAL = \mathbf{F}AVAL$$

The initialisation operation CAI is specifed by

$$AI = CAI; L$$

This equation can determine a function CAI by virtue of the formula:

$$\{s\}\rceil CAI = \{\{s\}\rceil AI\}$$

for all global data s. Moreover it is strightforward to show that for all global data s

$$\begin{array}{rcll} \{s\}\rceil AI & = & \{\{s\}\rceil AI\}\rceil L & \textit{def of } L \\ & = & (\{s\}\rceil CAI)\rceil L & \textit{def of } CAI \\ & = & \{s\}\rceil (CAI; L) & \textit{by law (6) in appendix} \end{array}$$

i.e., CAI really satisfies the given equation.

The finalisation operation CAF is defined by

$$CAF = L; AF$$

Finally, for each index $i \in \mathbf{I}$ the operation $caop_i$ is specifed by the equation

$$caop_i; L = L; aop_i$$

The existence of a deterministic solution $caop_i$ is obvious since $caop_i$ can be defined in the similar way as CAI.

Now we have a canonical data type **CA**, and can show

Theorem 8. **CA** refines **A** by the upward simulation L, and for all P in $\mathcal{D}$

$$P_{\mathbf{A}} = P_{\mathbf{CA}}$$

Proof: Direct from the definition L and **CA**, and lemma 8 in appendix. □

We are now ready for the main theorem of this section.

Theorem 9. If **C** refines **A** then there are an upward simulation L from **CA** to **A**, and a downward simulation from **CA** to **C**.
Proof: If **C** refines **A**, **C** thus refines **CA** by the fact that $P_{\mathbf{A}} = P_{\mathbf{CA}}$. By applying theorem 6, we can find a downward simulation from **CA** to **A**. This completes the proof. □

5 Conclusion

We have introduced two simulation conditions which guarantee that a concrete data type refines an abstract one. These simulation conditions are more general than the rule used in VDM: the downward rule always applies if the VDM rule does, but there are situations to which the downward rule applies though the VDM does not. In cases where botb rules apply, the VDM relation is total and surjective though the downward simulation need not to be; when the downward simulation is a bijection, the two rules coincide.

The simulation relations recommended in section 3 can be used not only in treatment of the total correctness of a design, but also in the derivation of a concrete data type from an abstract data type. The effective way of using the the result of this paper is as follows

(1) First design and maybe use the abstract type **A**

(2) Choose some suitable simulation relation R

(3) Calculate the weakest specification of concrete data type as follows:

$$\begin{aligned} CI &= AI; R \\ CF &= AF/R \\ cop_i &= (aop_i; R)/R \qquad \textit{for each index } i \in \mathbf{I} \end{aligned}$$

or

$$\begin{aligned} CI &= R\backslash AI \\ CF &= R; AF \\ cop_i &= R\backslash(R; aop_i) \qquad \textit{for each index } i \in \mathbf{I} \end{aligned}$$

(4) Check that the domain of the concrete operations are weak enough (for example, total commands are still total).

The use of calculation in step (3) is a promising innovation. If **A** is an abstract operational semantics of a programming language, the method may be useful in deriving the concrete machine code to be produced by a compiler for a concrete machine.

One problem in this paper is that refinement for a restricted language does not imply refinement for the more general language, which might have more powerful tests to discriminate data types. So, although the methods described in this paper are perfectly valid, they might not strong enough to prove every refinement in more powerful languages. This problem is investigated in [6].

Acknowledgement

To many members of the Programming Research Group for helpful advice and suggestions

of varions kinds. The research is supported by the Science and Engineering Research Council of Great Britain.

References

[1] E.W. Dijkstra, *A Discipline of Programming*, Prentice-hall, Englewood Cliffs, NJ, 1976.

[2] D. Gries and J. Prins, *A New Notion of Encapsulation*, SIGPLAN Notices 20 (7) (1985) 131-139.

[3] He, Jifeng, C.A.R. Hoare and J.W. Sanders, *Data Refinement Refined*, (Resume) LNCS 213, (1986) 187-196.

[4] C.A.R. Hoare, *Proof of Correctness of Data Representation*, Acta Informatica 1 (1972) 271-281.

[5] C.A.R. Hoare and He, Jifeng, *The Weakest Prespecification*, Inform. Process. Lett. 24 (2) (1987) 127-132.

[6] C.A.R. Hoare and He, Jifeng, *Data Refinement in Categorical Setting*, to appear.

[7] C.B. Jones, *Software Devclopment: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[8] T. Nipkow, *Nondeterminstic Data Type*, Acta Informatica 22 (1986) 629-661.

Appendix

The following laws presented in [5] will be used in the later proofs.

(1) If b is a condition, that is b;U=b, then
$P;(b \cap Q) = (P \cap \breve{b});Q$

(2) $(P;U) \cap I_S \subseteq P;\breve{P}$

(3) If f is a partial function then $\breve{f};f \subseteq I_S$

(4) let $\kappa = \{\perp\} \times S$ where $\perp$ denotes the undefined state.
Then $U \backslash \kappa = \kappa$

(5) $S]P = S$ *if* $\kappa \subseteq P$

(6) $B](P;Q) = (B]P)]Q$

(7) $\{s\}](\bigcap_i P_i) = \bigcap_i (s]P_i)$

(8) $(P;Q)\backslash R = P\backslash(Q\backslash R)$

Definition. For any relation P we define

$$domP = \{\rho \mid \exists \tau.\rho P\tau \wedge \neg \rho P \perp\}$$

and $ranP = S]P$

Lemma 1. If R is a downward simulation from A to C then for all $P(\mathbf{A})$ in $\mathcal{D}(\mathbf{A})$

$$R;P(\mathbf{C}) \subseteq P(\mathbf{A});R$$

Proof: The proof is based on structural induction.

(a). Base case, let $P = X$. For each index $i \in I$

$$\begin{array}{lcll} R;P(cop_i) & = & R;cop_i & def\ of\ P \\ & \subseteq & aop_i;R & def\ of\ downward\ simulation \\ & = & P(aop_i);R & def\ of\ P \end{array}$$

When $P = I$ or $P = U$, the conclusion is obvious.

(b). Assume that

$$R;P(\mathbf{C}) \subseteq P(\mathbf{A});R \ \textit{and}\ R;Q(\mathbf{C}) \subseteq Q(\mathbf{A});R$$

Then it is easy to conclude that

$$\begin{array}{lcll} R;(P(\mathbf{C});Q(\mathbf{C})) & = & (R;P(\mathbf{C}));Q(\mathbf{C}) & the\ associativity\ of\ ; \\ & \subseteq & P(\mathbf{A});R;Q(\mathbf{C}) & by\ the\ assumption \\ & \subseteq & P(\mathbf{A});Q(\mathbf{A});R & by\ the\ assumption \end{array}$$

$$
\begin{array}{rcll}
R;(P(\mathbf{C}) \cup Q(\mathbf{C})) & = & R;P(\mathbf{C}) \cup R;Q(\mathbf{C}) & ;\ \textit{distribute through}\ \cup \\
& \subseteq & P(\mathbf{A});R \cup Q(\mathbf{A});R & \textit{by the assumption} \\
& \subseteq & (P(\mathbf{A}) \cup Q(\mathbf{A}));R & ;\ \textit{distribute through}\ \cup
\end{array}
$$

(c). Assume that for all $n \geq 0$ $P_n \supseteq P_{n+1}$ and $R;P_n(\mathbf{C}) \subseteq P_n(\mathbf{A});R$

then we have

$$
\begin{array}{rcll}
R;(\bigcap_n P_n(\mathbf{C})) & \subseteq & \bigcap_n (R;P_n(\mathbf{C})) & \textit{the monotonicity of}\ ; \\
& \subseteq & \bigcap_n (P_n(\mathbf{A});R) & \textit{by the assumption} \\
& \subseteq & (\bigcap_n P_n(\mathbf{A}));R & \textit{the cocontinuiity of}\ ; \ \square
\end{array}
$$

Lemma 2. $P;(Q \setminus R) \subseteq Q\backslash(P;R)$
Proof:

$$
\begin{array}{rcll}
LHS;Q & = & P;(Q\backslash R;Q) & \textit{the associativity of}\ ; \\
& \subseteq & P;R & \textit{def of}\ \setminus \\
LHS \subseteq RHS & & & \textit{def of}\ \setminus \qquad \square
\end{array}
$$

Lemma 3. If f is a partial function, and $Q;U = U$, then

$$Q\backslash(f;P) = f;(Q\backslash P)$$

Proof:

$$
\begin{array}{rcll}
 & X;Q \subseteq f;P & & \\
\Rightarrow & X;Q;U \subseteq f;P;U & & \textit{the monotonicity of}\ ; \\
\Rightarrow & X;U \subseteq f;U & & Q;U = U\ \textit{and}\ P;U \subseteq U
\end{array}
$$

Moreover we have

$$
\begin{array}{rcll}
 & X;Q \subseteq f;P & & \\
\Rightarrow & \breve{f};X;Q \subseteq P & & \textit{by law}\ (3) \\
\Rightarrow & \breve{f};X \subseteq Q\backslash P & & \textit{by def of}\ \setminus \\
\Rightarrow & (f;U \cap I_S);X \subseteq f;(Q\backslash P) & & \textit{by law}\ (2) \\
\Rightarrow & (f;U \cap X) \subseteq f;(Q\backslash P) & & \textit{by law}\ (1) \\
\Rightarrow & X \subseteq f;(Q\backslash P) & & X \subseteq X;U\ \textit{and}\ X;U \subseteq f;U
\end{array}
$$

which implies that

$$Q\backslash(f;P) \subseteq f;(Q\backslash P)$$

From lemma 2 it follows that

$$f;(Q\backslash P) \subseteq Q\backslash(f;P)$$

which leads to the conclusion. $\square$

Lemma 4. If P is a total finitary relation, and $\{Q_i\}$ is a descending chain satisfying for all $i \geq 0$ $k \subseteq Q_i$ then

$$P;(\bigcap_i Q_i) = \bigcap_i (P;Q_i)$$

Proof: Here we dinstinguish two cases:

Case 1: $\{s\}\rceil P = S$

$$\begin{aligned}
\{s\}\rceil(P;\bigcap_i Q_i) &= S\rceil(\bigcap_i Q_i) && \textit{by law (6) and the assumption}\\
&= S && \textit{by law (5)}\\
&= \bigcap_i (S\rceil Q_i) && \textit{by law (5)}\\
&= \bigcap_i (\{s\}\rceil(P;Q_i)) && \textit{by law (6) and the assumption}\\
&= \{s\}\rceil \bigcap_i (P;Q_i) && \textit{by law (7)}
\end{aligned}$$

Case 2. $\{s\}\rceil P = \{t_0, ..., t_n\}$

$$\begin{aligned}
\{s\}\rceil(P;\bigcap_i Q_i) &= \bigcup_{j\leq n} \{t_j\}\rceil(\bigcap_i Q_i) && \textit{by law (6) and the assumption}\\
&= \bigcup_{j\leq n} (\bigcap_i \{t_j\}\rceil Q_i) && \textit{by law (7)}\\
&= \bigcap_i (\bigcup_{j\leq n} \{t_j\}\rceil Q_i) && \textit{finite union distributes through}\\
& && \textit{the intersection of a descending chain}\\
&= \bigcap_i (((\{s\}\rceil P)\rceil Q_i) && \textit{by the assumption}\\
&= \{s\}\rceil \bigcap_i (P;Q_i) && \textit{by law (7)}
\end{aligned}$$

□

Lemma 5. If f is a partial function, for all $i \geq 0$ $k \subseteq Q_i$ then

$$f;(\bigcap_i Q_i) = \bigcap_i (f;Q_i)$$

Proof: Similar to lemma 4. □

Lemma 6. For all P in $\mathcal{D}$

$$\kappa \subseteq R(P)$$

Proof.

$$\begin{aligned}
LHS &= U\backslash\kappa && \textit{by law (5)}\\
&\subseteq ((P(\mathbf{C});CF)\backslash(P(\mathbf{A});AF) && \textit{since } P(\mathbf{C});CF \subseteq U\\
& && \textit{and } P(\mathbf{A});AF \supseteq \kappa\\
&= RHS && \textit{def of } P
\end{aligned}$$

□

Lemma 7. If $\mathbf{C}$ refines a canonical data type $\mathbf{A}$, then for all $i \in I$

$$R;cop_i \subseteq aop_i ; R$$

$$
\begin{array}{lll}
 & R \subseteq (cop_i; P(\mathbf{C}); CF) \backslash (aop_i; P(\mathbf{A}); AF) & \textit{def of } R \\
\Rightarrow & R \subseteq aop_i; ((cop_i; P(\mathbf{C}); CF) \backslash (P(\mathbf{A}); AF)) & \\
 & & \textit{lemma 3 and since } \mathbf{A} \textit{ is canonical} \\
\Rightarrow & R \subseteq aop_i; (cop_i \backslash R(P)) & \textit{by law (8)} \\
\Rightarrow & R; cop_i \subseteq aop_i; (cop_i \backslash R(P)); cop_i & \textit{the monotonicity of ;} \\
\Rightarrow & R; cop_i \subseteq aop_i; R(P) & \textit{def of } \backslash \\
\Rightarrow & R; cop_i \subseteq \bigcap_{P \in D} (aop_i; R(P)) & \textit{set theory} \\
\Rightarrow & R; cop_i \subseteq aop_i; R & \textit{lemma 5 and lemma 6} \quad \square
\end{array}
$$

Lemma 8. For all P in D

$$P_{\mathbf{A}} = P_{\mathbf{CA}}$$

Proof: It is similar to lemma 1, and omitted. □