Aspects of Computation

William P. Coleman
MIEMSS, UMAB; Baltimore, MD 21201, *and*
Department of Mathematics, UMBC; Baltimore, MD 21228
coleman@umbc2.umbc.edu

Abstract

In this paper I try to look carefully at a few examples and to find suggestions toward a theory of computation that would be rich enough to represent distinct processes in distinct ways. As much as anything, it is a question of finding a terminology that can describe computational processes and that has content by being located within a formal mathematical structure. In particular, I look at processes in computing machines from the point of view that they must be interpreted as representing real-world processes.

1. Introduction.

Matthew Hennessy concludes his book [1] on the *Algebraic Theory of Processes* with the following:

> As this avenue of research is developed to include languages with more sophisticated features we will require a more sophisticated mathematical framework, and it is unlikely to be found in the existing literature.... As a second example, we could take the language in Hoare 1978 or the full version of **CCS** in Milner 1980. Here values are passed along the channels so that the actions are no longer uninterpreted. Rather, the port names now act as variable binders in the same way as λ-abstraction in the λ-calculus....

Computational and natural processes are very confusing: processes that appear similar, and even require abstractly similar mathematics, are "really" different. Worse, most useful processes are convoluted combinations of such easily mistakable processes. Thus, obscurities, such as the one Hennessy complains of, arise.

I shall try to contribute to clarification by considering a number of examples and analyzing them with some care. My analyses may seem almost metaphysical — in a way, they are. I shall suggest the rudiments of a comprehensive mathematical theory but shall not attempt the full-blown treatment that Hennessy calls for.

My point of view in this paper can be captured by the slogan: **The first, although not the last, requisite of a mathematical theory of computation is the ability to represent distinct behavior in distinct ways, and therefore the first job of the theorist is to build a formal language rich enough to do so.** Our immediate need is not parsimony but catholicity: Can we build a logic that can integrate and explain the valid insights of the various theories?

"*Reductivists need not apply.*"

2. Some Motivation.

2.1. Three Examples.

Example A A biochemist [2] wishes to study the metabolism of the amino acid leucine in the body. The plan is to infuse labeled leucine (molecules one of whose carbon atoms is a stable isotope that can be detected by mass spectrometry) into a subject at a constant rate, and then collect expired air at intervals to measure the "enrichment," the proportion of CO_2 molecules that are labeled. A simplified picture of the possible reactions is

$$\textbf{protein} \rightleftharpoons \textbf{leucine} \rightarrow \textbf{CO}_2. \qquad (2.1)$$

The biochemist would like to know the rate of incorporation of leucine into protein compared to the rate of catabolism of leucine into CO_2. □

Example B An electrical engineering laboratory has a system of independent software modules that it uses

This research was partially supported by a grant from the Life Sciences Division of the United States National Aeronautics and Space Administration (NASA).

to keep track of results. The module **ohm** takes input in the form,

```
/er/ e, r,
```

and uses Ohm's Law $i = e/r$ to compute the output,

```
/i/ i.
```

A second module **watt** computes

```
/w/ w
```

according to $w = e \times i$. Thus, for an experiment in which a succession of circuits are each measured once, one might use a shell script that follows **ohm** by **watt**.

Another way to think of this example is to imagine it, not implemented as software in a general-purpose computer, but hardwired in a pair of special devices. **Ohm** has a bank of two registers with LED readouts corresponding to **/er/**, and a single register corresponding to **/i/**. The operator sets **/er/**, presses a button, and **i** appears. There is a second device implementing **watt**, and its inputs are direct connections from **ohm**. □

Example C The protocol of a simple message-forwarding device is to accept a message, wait until the line is clear, and then forward it. Any other messages arriving while it is waiting are rejected. One models the history of the message as

$$\textbf{not forwarded} \rightarrow \textbf{forwarded}, \tag{2.2}$$

and the history of the device as

$$\textbf{ready} \underset{\textbf{s}}{\overset{\textbf{r}}{\rightleftarrows}} \textbf{busy} \tag{2.3}$$

It changes state in response to the rest of the system: it receives (**r**), or is enabled to send (**s**), a message. □

2.2. Instances.

The pictures, including the data structures, shown in these examples must be distinguished from the *instances* that can *populate* them.

There are two processes involved in example A. (1) A labeled carbon atom travels in time among the states in the set, {protein, leucine, CO_2}. It follows the *particle process,* which closely reflects picture 2.1 and whose probabilities are determined by numbers assigned to the arrows. (2) A measurement made by the biochemist has a state that is a number, the total number of particles that at one time are in a given state of the particle process. This measurement travels in time within a set of possible numbers. It follows the *population process,* whose probabilities are determined (i) by the particle process, (ii) by the process according to which particles arrive, and (iii) by the assumption that the particles have independent histories.

The crux of example A, as a stochastic process problem, is to show how measurements of the population process (which the biochemist is able to observe) allow one to infer properties of the particle process (which the biochemist wants to know). However, we shall not be concerned with the probabilistic aspects.

2.3. Programs and Devices: Processes and State Machines.

This background makes clearer the way in which examples A and B differ from C. The instances of B, like those of A, have independent histories: as each circuit is measured, its **e** and **r** are entered, and its **i** and then its **w** are computed without interference from those of any other circuit. However, in example C the histories of the messages are highly dependent.

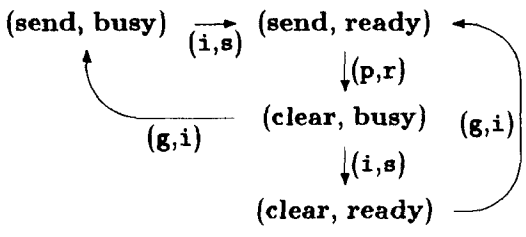
Pictures 2.1 and 2.3 seem similar, but they are not: 2.1, like 2.2, is a *process,* whereas 2.3 is a *state machine.* Their pictorial conventions are different: in 2.1 the nodes are inputs and the arrows are programs (chemical reactions), whereas in 2.3 the nodes are states and the arrows are inputs.

An instance of a state machine is a *device,* an object: a computer, perhaps, or even a person. A process is *implemented by* a computer program, and its instances are *runs* or *executions* of the program on a device.

These two representations — process and state machine — are complementary. A computation can be viewed as a process, in which case we prescind from the question of how its runs interact with those of other processes, or with other runs of the same process. Equally well, it can be viewed as a behavior of a state machine, in which case our model of the machine should show us how several runs can interact. The interest in the analysis of example B is in following how a single input **/er/** propagates from device to device across the system; thus it is more useful to depict it as a process. In example C, the interest is in seeing how a single device responds to a succession of inputs.

2.4. Levels of Granularity.

Think of picture 2.3 as showing the second of a pair of devices, $\mathcal{M}_1$ and $\mathcal{M}_2$, the first of which sends the message. They collectively form the system $\mathcal{M}$:

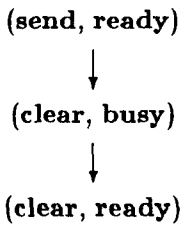


(2.4)

When all messages have been cleared from $\mathcal{M}_1$ and $\mathcal{M}_2$, then the system $\mathcal{M}$ is in the state (**clear, ready**).

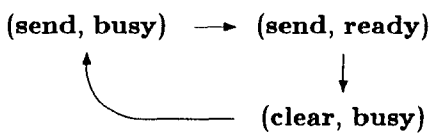
Generation of a new message by M_1 moves M to **(send, ready)**, and then to **(clear, busy)**, once the message is picked up by M_2. Then there are two possibilities. If the next event is that M_2 forwards the message, then M moves to back to **(clear, ready)**. If M_1 generates another message first, then M moves to **(send, busy)**, from which it moves back to **(send, ready)** after the first message is forwarded.

Several features of this example need clarification. I raise them here and hope that the mathematics of section 3 partly provides some explanation. M_1 and M_2 are *components* of M. Very roughly, one maps picture 2.4 to picture 2.3 by sending **(send, ready)** and **(clear, ready)** to **ready**, and sending **(clear, busy)** and **(send, busy)** to **busy**. On the other hand, the process shown in picture 2.2 is also related to certain fragments of the state machine in picture 2.4: to



, (2.5)

or to



, (2.6)

or to

(2.7)

Simplifying a process must imply a corresponding simplification in the machinery it runs on.

2.5. Types and States.

In example B, the module **ohm** computes a certain function from the set of states of **e** and **r** to the set of states of **i**:

$$ohm : E \times R \to I, \qquad ohm : (e, r) \mapsto i = e/r. \quad (2.8)$$

Note that *ohm* is a mathematical function, while **ohm** is a computer program, hard or soft. Also e, r, and i are (variable, or unspecified) numbers, ranging over the sets E, R, and I, while **e**, **r**, and **i** denote certain storage locations in a computer.

One usually identifies the sets E, R, and I with $\mathbf{R}_+$, the set of nonnegative real numbers, with which they are abstractly isomorphic. This is especially tempting in view of our moral desire to emphasize to the programmer the need to check his values. However, it is perhaps better to lecture the programmer at some other time, and to remember that the object of using a process to analyze example B is to be able to follow the sequence of program modules that are activated, and the sequence of outputs that they compute after the input of values **/er/** to the system. This is impossible unless we regard I as the set of values of a specific output of **ohm** and of a specific input to **watt**, and distinguish it from E, from R, and from $\mathbf{R}_+$.

I therefore, in defiance of tradition, define a *type* as the set of values of a specific input to, or output from, a computer program. The values themselves are *states*. The usual types, those of the programming languages, almost never meet this definition: **integer** is the input to some general-purpose device that implements a computer program, not to the program itself. (A theory that hopes eventually to describe parallel and distributed processing will have to distinguish programs from the devices they run on.)

2.6. Realization.

Example D A real-time monitoring device measures atmospheric pressure and collects data once per second, using a program module, **acq**, to acquire each data point. The output of **acq** is in the form,

/raw/ seq#, press. (2.9)

This is too dense a rate, so there is a second module, **avg**, that is enabled to run after n data points have been collected, and that averages them:

/averaged/ seq#, pressbar. (2.10)

The averaging interval,

/int/ n, (2.11)

can be set by the user.

There is a shell script that allows an execution of **avg** each time that **acq** has collected n points. Each execution of a module is an instance and has its own data, and these instances would be collected in a file.

int
n
2

raw	
seq#	press
acq1	131.
acq2	139.
acq3	135.
acq4	134.
acq5	138.

averaged	
seq#	pressbar
avg1	135.0
avg2	134.5

These data items might be created in the order: **acq1**, **acq2**, **avg1**, **acq3**, **acq4**, **avg2**, **acq5**.

Note that **press** and **pressbar** describe the outside world, whereas **n** describes the operation of the device. (More precisely, **n** contains control information that *prescribes* the operation of the device; it only *describes* it after a successful run.)

The value of **pressbar** does not have an independent meaning: it needs the value of **n** to complete it. □

In the computer, the data record /raw/ can have many instances, and unlike its counterpart in the outside world, several can exist in the computer simultaneously. Think of the computer as a complex, general-purpose, device made up of simpler devices, including logical units, floating-point units, and storage locations. As a device, the computer follows a certain state machine. The shell runs **avg** repeatedly to access different instances of /raw/ in different locations.

We can use the computer to *realize* such a shell script by associating its storage locations with instances of /raw/ and /averaged/ and associating appropriate sequences of machine states with **acq** and **avg**. Within an execution, **avg** is realized repeatedly, and differently: it runs on a different, but overlapping, set of locations.

2.7. Interpretation.

A theory of computation must provide for the fact that most computer processes deal with *data* that are to be interpreted as meaning something in the outside world.

In example D, data records are generated in the computer by instances of the shell script. Meanwhile, states — namely, atmospheric pressures — are generated in the outside world by instances of a meteorologic process, whose other details we ignore. The computer process is not homologous to the meteorologic one, but still the states of the storage locations /leucine/ are to be *interpreted* as describing the meteorologic states.

One has to map state to state and instance to instance. The structure of /raw/ is shown in picture 2.9. Different pieces contribute towards the interpretation in different ways. The name of the **press** field tells that the values found in any instance are of type *PRESS*. In fact, one might regard **press** as a function that maps the set of states of the /raw/ data records into the set *PRESS*. The convention of using the same name in a different typeface is meant to implement part of the interpretation by associating *PRESS* with a corresponding procedure for measuring the pressure, and thus with a state of the meteorologic process. The **seq#** field implements the rest of the interpretation: it maps an instance of /raw/ to an instance in which the pressure is measured. Altogether, these conventions make the data record assert that the value of **press** denotes the value measured on that occasion.

2.8. Simplification.

Example E A nurse at an intensive care unit workstation [2] [3] requests a pair of plots for each of two patients. They are provided by a system of independent software modules whose operation is highly nested. The subprocess **plotpts**, shared by several processes, plots a data pair **xval**, **yval**. It first computes the screen coordinates **xplot**, **yplot** actually to be plotted. These are given in terms of the previously established minimum and maximum x and y values **xmin**, **xmax**, **ymin**, **ymax** and the physical resolution **xpnts**, **ypnts** of the screen by the formulae,

$$xplot = \frac{xval - xmin}{xmax - xmin} \times xpts,$$

$$yplot = \frac{yval - ymin}{ymax - ymin} \times ypts.$$

Another module can then plot the point (*xplot*, *yplot*). The structure of the data is

```
/minmax/ patient#, plot#, xmin, xmax,
   ymin, ymax,
/screen/ patient#, plot#, xpnts,
   ypnts,
/vals/ patient#, plot#, xval, yval,
/plot/ patient#, plot#, xplot, yplot.
```

□

Abstractly, **plotpts** is a function — a black box — for computing /plot/ from /screen/, /minmax/ and /vals/. Note that these three inputs are bound at very different time scales within the total computation: /screen/ is a constant for that workstation, /minmax/ is a constant for each pair of plots, and /vals/ changes with each call to **plotpts**.

Some less ambitious group of programmers might have restricted their software to run only on screens of a fixed resolution, say 1024×1024. They would have written a module $\mathbf{plotpts}_{1024}$ with inputs /minmax/ and /vals/ and output /plot/; everything is the same, except that /screen/ is a fixed constant. In fact, for each such group of programmers and each fixed value

$$\mathbf{s} \in SCREEN = XPNTS \times YPNTS,$$

we have a different program $\mathbf{plotpts}_{\mathbf{s}}$. This suggests another way of looking at the original **plotpts**. We might look at a state **s** of /screen/, not as denoting a pair of numbers, but as denoting the program $\mathbf{plotpts}_{\mathbf{s}}$, or the process corresponding to it.

A user who confines himself to a 1024×1024 screen will notice no difference between the original set of modules and the same set with **plotpts** replaced by $\mathbf{plotpts}_{\mathbf{s}}$. We need a formal notion to capture the way in which a complicated process with many states compares to a *simplification:* a program that, within a restricted domain, performs the "same" computations.

2.9. Language.

We are now in a position to clarify the different senses in which data records are "generated."

History. Records like those in example D are generated in the computer in a historical sequence involving the execution of computer programs.

Syntax. The syntax of the **averaged** records in example D, is defined by the expression 2.10, and any record

must meet this specification in order to be correct. If this example were not so fragmentary, we might see rule 2.10 as part of a generative grammar of which the records are productions. There is no relation between the sense in which this grammar generates the records and the sense in which **avg** generates them, beyond the requirement that **avg** generate correct instances.

Semantics. The interpretation described in section 2.7 gives a meaning to the records /raw/.

2.10. Synchronization.

Example F There is a small computer that represents only natural numbers and has no hardware for addition, but it does have sophisticated control structures and can be programmed to add using the recursive definition,

$$n + 0 = n, \qquad n + (m + 1) = (n + m) + 1. \quad (2.12)$$

The shell script **plus** has modules implementing

$$succ : R \to R, \qquad succ : r \mapsto (r + 1); \quad (2.13)$$

$$pred : M \to M, \qquad pred : m \mapsto (m - 1). \quad (2.14)$$

It can also **copy** a number from one location to another. Its data structures are

/arg1/ n,
/arg2/ m,
/result/ r.

Plus says to run **pred** until **m** is in state 0, do **copy** once, and then run **succ** until reaching **r1**.

In the following illustration, the input data are $n = 3$ and $m = 2$, and the result is $r = 5$.

arg1	
seq#	n
n1	3
n2	3
n3	3

arg2	
seq#	m
m1	2
m2	1
m3	0

result	
seq#	r
r1	5
r2	4
r3	3

It runs as follows: load **n1** and **m1**; and then **pred**, **m2**; **pred**, **m3**; **copy**, **r3**; **succ**, **r2**; **succ**, **r1**; stop. □

In the introduction to his book [1], Hennessy presents a set of progressively more subtle and puzzling examples. In comparing two of them he says [p. 12], "However, to *justify* distinguishing them one needs a coherent view or understanding of nondeterminism." After considering other such pairs, he observes,

> Of course any particular model [of computational processes] decides between identifying and distinguishing such pairs. However, the justification lies hidden in the definition of the model or in the mind of the designer of the model. We argue that this justification which underlies the model is its most important feature and should always be elucidated.

The plea that one job of the model should be to make the features of the process explicit is very much in the spirit of the present paper. However, I would oppose the idea that there is a global theory of nondeterminism that justifies such choices.

Hennessy's examples, and mine as well, are designed to show that the interaction of subprocesses within a process may by very idiosyncratic. In example D, the shell alternates n runs of **acq** with a single run of **avg**, where n is a parameter set by the user. In example F, the shell runs **pred** a variable number of times, but here the condition for termination is that a location **m** reach a specified value, zero. (The shell has no way of determining, from general principles, that the number of runs equals the value of **m1**.) Then, after running **copy**, it runs **succ** a variable number of times, terminating when it has bound a value, any value, in the location **r1**. I think that this suggests that subprocesses can be combined by a very large number of possible combinators. The point of modeling is to illustrate these explicitly, and not to implement an abstract theory.

The question whether two processes are distinguishable is local, not global: tell me what windows you are planning to view them through (in the near future), and I shall tell you whether you can distinguish them by looking through those windows. A test of the goodness of a model is whether it has the means to make this promise good.

3. Some Mathematics.

3.1. Connections to the Literature.

The theory presented in the following sections is developed from that in [2] and [4]. It is, in different ways, modeled on category theory (especially topos theory), universal algebra, and symbolic logic. It parallels, in different places, the standard theories of computation found in [1], [5] and [6], and might be viewed as an attempt to integrate their insights into a single structure. The books [7] [8] are constantly in the background.

The standard reference for category theory is [9]. The books [10] and [11] are helpful introductions oriented towards computer scientists, while [12] is oriented towards logicians. Similarly, [13] is a helpful introduction to order theory and lattices, with examples from computer science. It would prepare one for [14].

3.2. Categories.

Picture 2.4 is an example of what category theorists call a *graph*. Such a graph **G** can be made into a *category* **C** by adding in an *identity arrow* from each node, or *object,* to itself; and for each consecutive pair f, g of arrows an arrow $g \circ f$, their *composition*. In this context, composition would mean consecutive operation of f and then g, and identity would be a null process.

Thus a category **C** consists of a set $\mathbf{C}_O$ of objects and a set $\mathbf{C}_A$ of arrows. (For legibility, one suppresses the A and the O, where possible.) It is required that the identities behave like identities and that the operation $\circ$ be associative. It is not required that each $g \circ f$ be distinct from the ones previously included, but if this is always so then **C** is $F\mathbf{G}$, the *free category on* **G**.

A different example of a category is **Set**, whose objects are small sets and whose arrows are functions between the sets.

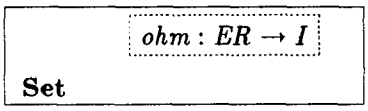
3.3. Actions.

Processes. For example B, we could start with the graph **G**:

$$/\mathtt{er}/ \xrightarrow{\mathbf{ohm}} /\mathtt{i}/. \tag{3.1}$$

To get a category **C** based on **G**, we would add in the identity arrows $1_{/\mathtt{er}/}$ and $1_{/\mathtt{i}/}$. Then, since, in this example, the only possible compositions are $\mathbf{ohm} \circ 1_{/\mathtt{er}/} = \mathbf{ohm}$ and $1_{/\mathtt{er}/} \circ \mathbf{ohm} = \mathbf{ohm}$, we would be done.

This would give a good idea of the possible sequences of program activation, but it fails to capture the meaning given by picture 2.8. The point is that **ohm** not only executes, but as it executes it computes an output dependent on the value of the input. Picture 2.8 is really a fragment of **Set**:



(3.2)

The way to pick out the particular fragment of **Set** that we need is to define a functor

$$C : \mathbf{C} \to \mathbf{Set}.$$

A *functor* is a mapping between categories; it has to send objects to objects, and arrows to arrows, in such a way that identities and composition are preserved. It has an *object function* $C_O : \mathbf{C}_O \to \mathbf{Set}_O$, and an *arrow function* $C_A : \mathbf{C}_A \to \mathbf{Set}_A$. Let $C(/\mathtt{er}/) = ER$, let $C(/\mathtt{i}/) = I$, and let $C(\mathbf{ohm}) = ohm$.

Such a functor expresses the *action* of **C** on **Set**.

State Machines. State machines can also be expressed as such actions.

For example C, let $\mathbf{G}_2$ be the picture 2.3 of device $\mathcal{M}_2$. Form $\mathbf{C}_2$ by adding in two identity arrows, and letting $\mathtt{r} \circ \mathtt{s} = 1_{\mathbf{busy}}$ and $\mathtt{s} \circ \mathtt{r} = 1_{\mathbf{ready}}$. The *action* of $\mathbf{C}_2$ is given by a functor $C_2 : \mathbf{C}_2 \to \mathbf{Set}$, defined as follows. Let Q_2 be the set,

$$Q_2 = \{\mathbf{ready}, \mathbf{busy}\}.$$

Q_2 is the image of both states: $C_2(\mathbf{ready}) = C_2(\mathbf{busy}) = Q_2$. Next, $C_2(\mathtt{r})$ and $C_2(\mathtt{s})$ are functions $Q_2 \to Q_2$:

$q \in Q_2$	$(C_2(\mathtt{r}))(q)$	$(C_2(\mathtt{s}))(q)$
ready	**busy**	**ready**
busy	**busy**	**ready**

3.4. Simplification.

Let

$$C : \mathbf{C} \to \mathbf{Set}$$

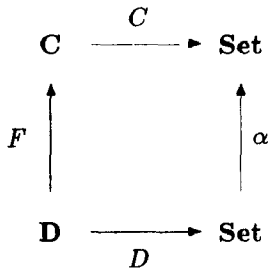
and

$$D : \mathbf{D} \to \mathbf{Set}$$

be two such actions. We capture the ideas of section 2.8 by saying that D *is a simplification of* C, and writing $D \leq C$, if we have the following. There is a functor $F : \mathbf{D} \to \mathbf{C}$. (It is not to be confused with the *free functor* F of section 3.2.) There is also a *natural transformation* $\alpha : D \xrightarrow{\cdot} CF$. This α is a collection of functions $\alpha_{\mathbf{d}}$, one for each object **d** of **D**, that go from the set $D(\mathbf{d})$ to the set $CF(\mathbf{d})$. These functions must consistently transform D into CF. That is, if $\mathbf{f} : \mathbf{d}_1 \to \mathbf{d}_2$ is an arrow of **D**, we must always have

$$\alpha_{\mathbf{d}_2} \circ D(\mathbf{f}) = FC(\mathbf{f}) \circ \alpha_{\mathbf{d}_1}.$$

Thus a simplification looks like the diagram,



The notation $\leq$ is justified by the fact that simplification is reflexive and transitive.

For example, let **C** be a category based on picture 2.4, and let C express its action on **Set**. Let C_2 and $\mathbf{C}_2$ be as in section 3.3. Then $C_2 \leq C$. The functor $F_2 : \mathbf{C}_2 \to \mathbf{C}$ has $F_2(\mathbf{ready}) = (\mathbf{clear}, \mathbf{ready})$; $F_2(\mathbf{busy}) = (\mathbf{clear}, \mathbf{busy})$; $F_2(\mathtt{r}) = (\mathtt{p}, \mathtt{r}) \circ (\mathtt{g}, \mathtt{i})$; and $F_2(\mathtt{s}) = (\mathtt{i}, \mathtt{s})$. The natural transformation α is then the corresponding function $Q_2 \to Q$.

We also, in this example, have $C \leq C_2$. Let $F(\cdot, \mathbf{ready}) = (\mathbf{ready})$; $F(\cdot, \mathbf{busy}) = (\mathbf{busy})$; $F(\cdot, \mathtt{r}) = (\mathtt{r})$; $F(\cdot, \mathtt{s}) = (\mathtt{s})$; and $F(\cdot, \mathtt{i}) = (\mathtt{i})$. What makes $\mathcal{M}_2$ be a *component* of $\mathcal{M}$ is the fact that $F \circ F_2 : \mathbf{C}_2 \to \mathbf{C}_2$ is the identity functor.

3.5. Language and Interpretation.

In section 2.9, I pointed out that data records are generated, in different ways, historically and syntactically. The semantic meaning is also generative, in sense that is hidden inside the requirement that the interpretation map the set of states of the data records to a set of states of the external process that they denote.

In section 2.8, we saw that it is natural to think of processes whose states are themselves processes. Conversely, states, and even types, can be thought of as very restricted processes: they are simplifications, in the sense of section 3.4, of the process in which they occur. Thus, we tend to blur the distinctions between processes, types, and states.

An object outside the computer might be capable of supporting a set a of processes. Let $\downarrow a$ be the system consisting of the processes in a and of all their simplifications. There will usually be a mutual dependency among the members of $\downarrow a$: for any two elements, one might be a simplification of the other; or they might both be simplifications of some third process; or some third process might be a simplification of both. Thus $\downarrow a$ is structured. Now look at a set A of data records in the computer whose members are mapped by an interpretation into members of $\downarrow a$. The members of A acquire thereby a semantic interdependency [3].

As in section 2.7, let a be a set of processes in the outside world, and let S be a set of objects. Let A be a set of processes in the computer. We want to view a data record of A as a declarative sentence constructed from the vocabulary a and S. For example, an instance of `/raw/` might be

$$\langle \mathbf{6144}, \mathbf{29.87} \rangle. \qquad (3.3)$$

It means that on the 6144th measurement, the geographic location measured had an atmospheric pressure of 29.87. As discussed in sections 2.6 and 2.7, these four pieces — 6144, the location, pressure, and 29.87 — are instrumental in giving record 3.3 its interpretation.

Section 2.6: In a realization, the state of a data record corresponds to a state of a type of a process in A. Section 2.7: We require a set of functions, each of which maps a type of A to an element of $\downarrow a$. (This takes care of **29.87**.) By composing with the realization, we also have, for each kind of data record a function mapping each field of the record to an element of $\downarrow a$. (This takes care of **pressure**.) Then, we require a function mapping each data record to an element of S. (This takes care of **location**, which is not explicitly indicated in the program, since only one location is in question.) Finally we require a function mapping each data record to an instance in which the process in $\downarrow a$ that corresponds to that record type runs on that individual in S. (This maps **6144** to a measurement.)

References

[1] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, MA, 1988.

[2] William P. Coleman. Computational logic of network processes. In Martin D. Fraser, editor, *Advances in Control Networks and Large Scale Parallel Distributed Processing Models*, Ablex Publishing Company, 1990. In press.

[3] William P. Coleman, David P. Sanford, and Andrea De Gaetano. Syntax and semantics of languages for medical information processing. In Søren Buus, editor, *15th Annual Northeast Bioengineering Conference*, pages 189–190, IEEE, 1989.

[4] William P. Coleman. Models of computational processes. *Journal of Symbolic Logic*, 55(1):437, March 1990. (Abstract of talk presented at Spring Meeting of the Association for Symbolic Logic, 1989. *Full paper in progress.*).

[5] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[6] Wolfgang Reisig. *Petri Nets.* Springer-Verlag, Berlin, 1985.

[7] Samuel Eilenberg. *Automata, Languages, and Machines.* Volume A, Academic Press, New York and London, 1974.

[8] Samuel Eilenberg. *Automata, Languages, and Machines.* Volume B, Academic Press, New York and London, 1976.

[9] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[10] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice Hall, New York, 1990.

[11] Peter Freyd and Andre Ščedrov. *Categories, Allegories.* North Holland, Amsterdam, 1990.

[12] Robert Goldblatt. *Topoi: The Categorial Analysis of Logic.* North-Holland Publishing Company, Amsterdam, revised edition, 1984.

[13] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, UK, 1990.

[14] Ralph N. McKenzie, George F. McNulty, and Walter F. Taylor. *Algebras, Lattices, and Varieties.* Volume I, Wadsworth and Brooks Cole, Monterey, CA, 1987.