CHAPTER 3

MATHEMATICAL FOUNDATIONS

The need to be sure that the semantics of a language are precisely understood applies to the meta-language. Indeed, if the meta-language is not adequately defined, all attempts to employ it may be wasted. Chapter 2 uses some extremely powerful concepts. In particular, recursive functions defined in Lambda notation and the self-application of functions need detailed mathematical foundations. These foundations were, in fact, lacking when Christopher Strachey and Peter Landin first used Lambda notation to define programming languages. The mathematical problems were first solved by Dana Scott. Scott has gone on to offer several mathematical models: the treatment here gives an outline of the recent "neighborhood" approach. A pedagogic treatment of the general problem can be found in [Stoy 77a].

Readers who are less interested in mathematics than their use in defining semantics, might choose to omit reading this chapter. If they are concerned with defining systems where there is no recursion, this could be done quite safely. Readers involved in the definition of programming languages should, however, be aware of the problems involved and are recommended to read at least the first two sections.

CONTENTS

3.1 INTRODUCTION

The VDM meta-language, presented in this book, is intended to be used as a vehicle for rigorous mathematical reasoning about systems specified with it. In this chapter we aim to examine some of the problems that seem to arise when we use it for this purpose, and to indicate how they may be resolved.

It might be hoped that this kind of study would play a subsidiary part in a course of training in the use of VDM. After all, a knowledge of the formal construction of the real numbers by Dedekind cuts is hardly essential before undertaking everyday arithmetical calculations. For this hope to be fulfilled we must provide some kind of guarantee that every grammatical construct in the notation means something sensible in the mathematical framework on which the notation is based -- or, at the very least, some simple rules of thumb about what to avoid (such as division by zero in the real number example). For the most part we shall in fact be able to give the appropriate guarantees, so that someone who writes a specification in this language can be assured that he is indeed specifying something -though not necessarily what he intended, of course.

The foundational work on which the apparatus of denotational semantics rests was principally done by Dana Scott. The present chapter will be largely devoted to an introduction to his theory: for a fuller account readers must be referred to [Scott 81a].

3.2 BASIC PROBLEMS

3.2.1 Circular Definitions

The first problem we must face concerns circular definitions. These arise not only in the semantics of recursive procedures (or data structures) but also for any constructs which involve looping. For example, we may wish to define the *while*-loop in such a way that

$$\text{"}\underline{\textit{while}}\ B\ \underline{\textit{do}}\ S\text{"} = \text{"}\underline{\textit{if}}\ B\ \underline{\textit{then}}\ (\ S;\ \underline{\textit{while}}\ B\ \underline{\textit{do}}\ S\ )\text{"} \qquad (1)$$

In the VDM language this same idea might be expressed (as in section 2 of chapter 2) as follows

$$M[mk\text{-}While(b,s)](state) \triangleq \quad (2)$$
$$(\underline{let}\ wh = \lambda\sigma.(\underline{let}\ bv=MX[b](\sigma)\ \underline{in}\ \underline{if}\ bv\ \underline{then}\ (M[s](\sigma))\ \underline{else}\ \sigma)\ \underline{in}$$
$$wh(state)$$

(Notice that (2) uses Church's λ-notation [Church 41a], which was also described in chapter 2.)

Example (1), since it is circular, is not *a priori* a definition at all. It is in effect an equation which we must try to solve for the value of

"*while B do S*"

and similarly in (2) the circular "definition" of the function $wh(\sigma)$ is an equation which we must try to solve for wh. However, the mere fact that we can write an equation does not guarantee that it has a solution: for example, in arithmetic the equation

$$x = x + 1$$

has none. Nor need any solution be unique; consider

$$x = x$$

Or, to take another example, consider the following recursively defined function.

$$f(x) = \underline{if}\ x=0\ \underline{then}\ 1\ \underline{else}\ \underline{if}\ x=1\ \underline{then}\ f(3)\ \underline{else}\ f(x-2) \quad (3)$$

We have to solve this for f. The solution which would naturally occur to most computer scientists is

$$f(x) = \begin{cases} 1 \text{ if } x \text{ is even and } x \geq 0 \\ \text{undefined otherwise} \end{cases} \quad (4)$$

But another solution is

$$f(x) = 1 \quad (\text{for all } x) \quad (5)$$

and so is

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is even and } x \geq 0 \\ a & \text{if } x \text{ is odd and } x > 0 \\ b & \text{otherwise} \end{cases} \qquad (6)$$

for any values of a and b. Thus part of the job of our theory is to guarantee the existence of solutions to any such circular equational definitions we need to write, and to tell us which solution is to be understood as the meaning of the "definition" when the solution is not unique.

3.2.2 Fixed Points

Using λ-notation we may rewrite our equations in the form

$$f = H(f)$$

where H is a λ-expression. For example, (3) might now become

$$f = H(f) \qquad (7)$$

where

$$H = \lambda g.\lambda x.\underline{if}\ x=0\ \underline{then}\ 1\ \underline{else}\ \underline{if}\ x=1\ \underline{then}\ g(3)\ \underline{else}\ g(x-2)$$

Now our search for a solution of the equation becomes a search for a fixed point of H; i.e., a value which is mapped by H to the same value. In most of our work these values will be functions themselves (notice that H above is a function of a function), but simpler functions may have fixed points too.

Examples of Fixed Points

Considering integers and functions on them:

- a fixed point of $\lambda x.x$ is x
- a fixed point of $\lambda x.8-x$ is 4
- a fixed point of $\lambda x.x$ is any integer
- a fixed point of $\lambda x.x+1$ does not exist.

Considering more general functions:

fixed points of $\lambda g.\lambda x.\underline{if}\ x=0\ \underline{then}\ 1\ \underline{else}\ \underline{if}\ x=1\ \underline{then}\ g(3)\ \underline{else}\ g(x-2)$

are the functions defined in (4), (5) and (6).

Fig. 1 Examples of Fixed Points

3.2.3 Self-Application and the Existence of Domains

The other main problem which we wish our theory to settle for us concerns the existence of the various value spaces we might define. Some of these seem unlikely. For example, consider the function (introduced in chapter 2)

$$twice(f) \triangleq \lambda x.f(f(x))$$

The argument of *twice* is itself a function; thus, for example *(twice square)3 = square(square(3)) = 81*. So far so good. But what are we to make of

$$((twice\ twice)square)3?$$

Obviously (at one level) this works out to be

$$\begin{aligned}&((\lambda x.(twice(twice\ x)))square)3\\ &= (twice(twice\ square))3\\ &= (twice\ square)((twice\ square)3)\\ &= 81^{**4} = 43046721\end{aligned}$$

But notice that in this calculation the function *twice* was applied to itself. Another example of such self-application is afforded by the following definition of the factorial function

$$\begin{aligned}a(b,x) &\triangleq \underline{if}\ x=0\ \underline{then}\ 1\ \underline{else}\ x \times b(b,x-1)\\ Factorial(y) &\triangleq a(a,y)\end{aligned}$$

Notice carefully that this involves no recursion: *a* is defined solely in terms of its parameters, and *Factorial* is defined solely in terms of its parameter *y* and the previously defined *a*; *a*, however, is required to take itself as an argument.

This kind of example is quite permissble in many programming languages. For example, chapter 2 mentions

$$PROCDEN = (VALUE \mid PROCDEN)^* \rightarrow STATE \rightarrow STATE$$

and remarks that the arguments in some particular *Argument-list* which corresponds to procedure parameters must be elements of *PROCDEN* itself.

Other languages might allow commands to be stored. A command is a value in the domain

$$STATE \to STATE$$

and

$$STATE = LOC \to VAL$$

where *VAL* is the domain of storable values, here including commands. So once again we have a circular domain definition.

Why should the circular definition of domains present any more of a problem than the circular definition of functions, already discussed? Let us consider a simpler case. Let F be a set of functions whose domain is G, another set of functions, and whose range is the set with the two elements 0 and 1. So

$$F = G \to \{0,1\}$$

Now if G has n elements it is easy to see that F will have 2^n elements: that is to say, the number of elements in F will always be greater than the number in G, and this remains so even when G contains *infinitely* many elements (this is Cantor's theorem). So even in this simple case there is no space of functions F such that

$$F = F \to \{0,1\}.$$

If the right hand side of this equation denotes all the functions from F to $\{0,1\}$ it will necessarily have too many elements.

Until we can resolve these difficulties we have no right to use any circular definitions, of functions or of domains, in our discussions. This would very much restrict our treatment of programming languages, but it would be necessary, in order to avoid the risk that we were talking nonsence by referring to things that could not possibly exist. Fortunately, however, these difficulties can be resolved, and in the remainder of this chapter we shall outline how this may be done.

Strictly speaking, perhaps, our investigation should proceed in two phases. First we should work out what is the minimal set of properties our system should have in order for the things we wish to do with it to be possible. Then secondly we should see if we could construct a system (a

"model") satisfying all these properties. That would avoid the danger of ending up with a system which had unnecessary constraints. Here, however, we shall begin inventing a model at once - taking care, though, that all the assumptions we make are reasonable requirements for a theory of computations.

3.3 INFINITARY OBJECTS

Functions provide one example of a class of infinitary objects, objects which can contain an infinite amount of information -- in this case the mapping for every element of an infinite domain. Infinitary objects cannot be handled explicitly within a finite machine. Instead we have to be satisfied, on each occasion, with a finite approximation to the object which will nevertheless be adequate for that particular occasion. The same situation arises with other infinitary objects, such as real numbers. We cannot write down π completely, because it would go on for ever; but we can always choose a finite approximation (25 decimal places, say) good enough for some particular occasion. All we can actually do with a function (that is, a mapping) is apply it to various arguments; and in a finite time we shall only be able to apply it to finitely many arguments. So in any particular execution of a program our knowledge of a function will be confined to the results it has given for just one particular finite set of arguments - that one particular finite approximation to the function is all that is relevant to that particular occasion. (We do not know in advance, of course, which particular subset we shall require for any particular execution -- that is why we represent the function by an algorithm, a procedure capable of generating any subset, though only ever a subset, as required.)

This is the general idea that we shall exploit: a theory about computing with infinitary objects must handle them as the limits of sets of finite approximations, so that in any particular computation one of these approximations will be adequate in itself. We now embark on the construction of such a system of elements.

3.3.1 Neighborhood Systems

We shall regard an approximation to an object as specifying the attributes that the object might actually possess. So let Δ be a set of such attributes, which we shall call *tokens*. We shall not have to worry

too much about what atom of information a single token represents: we shall only be concerned with collections of such tokens, subsets of Δ.

A neighborhood system over Δ is the family of those subsets of Δ which might possibly represent approximations to our objects. A particular member of this family, a neighborhood can be thought of as containing just those attributes which are not yet ruled out, and which might still apply to the object -- it might perhaps be thought of as a region of "possibility space".

There are two constraints on this family. In the first place, before any any computation has taken place no tokens will have been ruled out, so the whole set Δ is itself a neighborhood. Secondly, consider a pair of neighborhoods X and Y. These might, of course, be approximations of two quite different objects, with nothing in common; alternatively it might be possible to regard them as two approximations of the same object. This will certainly be possible if there is a third neighborhood, Z, in the system such that $Z \subseteq X$ and $Z \subseteq Y$. In this case, then, we insist that there be a neighborhood corresponding precisely to the information contained in just X and Y taken together, since X and Y together rule out all tokens which are not in their intersection. So we formally state

Definition 1: A family D of subsets of a given set Δ is called a neighborhood system over Δ if

1. $\Delta \in D$;
2. whenever $X, Y, Z \in D$ and $Z \subseteq X \cap Y$, then $X \cap Y \in D$

Notice that a smaller neighborhood represents a better approximation.

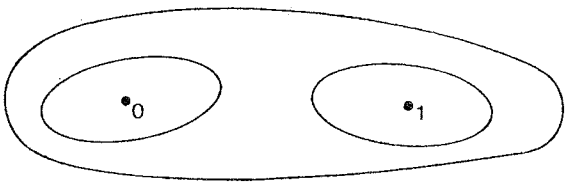

Fig. 2

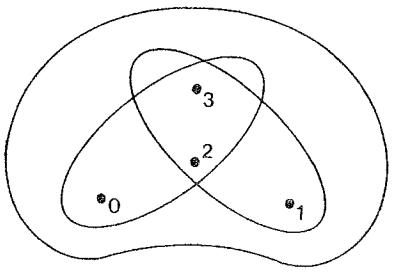

Fig. 3

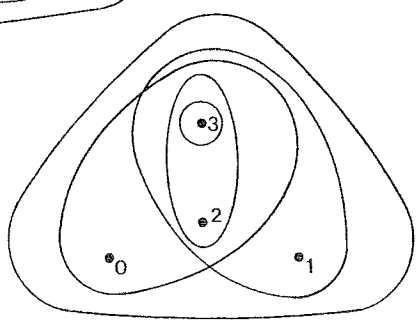

Fig. 4

Let us look at two simple examples. In the first (Fig. 2) there are two tokens and three neighborhoods. Either we have no information, or one of the tokens has been ruled out. In the next (Fig. 3) we have added two more tokens; notice that this does not require us to add any more neighborhoods -- the new system has precisely the same configuration of neighborhoods (under the set inclusion relation) as the previous one. However, if we choose to add neighborhood {3} to the system, the rules force use to add {2,3} too, and we have the system shown in Fig.4.

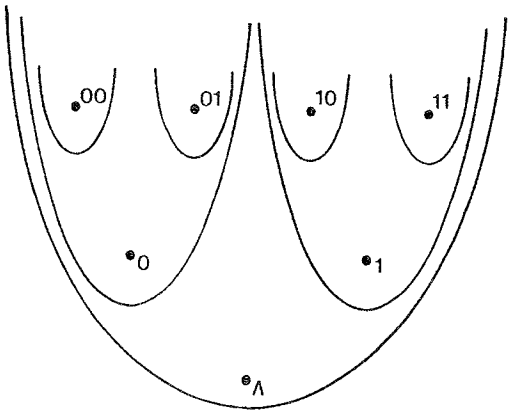

Fig. 5

In the other example (Fig. 5), the completely determined objects correspond to sequences of length two made up of zeroes and ones. The neighborhood labelled A, for example, encapsulates the approximation that the sequence begins with a zero.

3.3.2 Elements

We stated earlier that an element was to be regarded as the limit of a set of approximations. We now formally identify an element with the subfamily consisting of all those neighborhoods which could correspond to its approximations. That is to say, we are never going to be concerned about which particular sequence, of better and better approximations, is chosen from this sub-family. More precisely we have

Definition 2: The elements of a neighborhood system D are those subfamilies $x \subseteq D$ where

1. $\Lambda \in x$;
2. $X \in x$ and $Y \in x$ implies $X \cap Y \in x$,
3. whenever $X \in x$ and $X \subseteq Y \in D$, then $Y \in x$.

The third of these conditions, for example, says that whenever any particular neighborhood is a member any neighborhood corresponding to a worse approximation will also be a member. The collection of all the elements of D is written as $|D|$ and is known as a domain.

In the first of our examples (Fig. 2) there are three elements, as follows

{{0,1}}, {{0,1},{1}}, {{0,1},{0}}.

The first modification of this (Fig.3) similarly has just three elements:

{{0,1,2,3}}, {{0,1,2,3},{0,2,3}}, {{0,1,2,3},{1,2,3}};

the other version, however, with its extra neighborhoods (Fig.4), has extra elements as follows:

{{0,1,2,3}}, {{0,1,2,3},{0,2,3}}, {{0,1,2,3},{1,2,3}}
{{0,1,2,3},{0,2,3},{1,2,3},{2,3}},
{{0,1,2,3},{0,2,3},{1,2,3},{2,3},{3}}

In the other example there is an element corresponding to each of the tokens; it consists of all the neighborhoods containing that token.

3.3.3 Finite Elements

Each neighborhood of any neighborhood system determines a partial element, containing that neighborhood and all those corresponding to worse approximations. More precisely

Definition 3: For $X \epsilon D$ the element *upto* X is defined by

$$\textit{upto}\ X = \{\ Y \mid Y \epsilon D \wedge X \subseteq Y\ \}$$

These elements are called the finite elements of the domain $|D|$. The remaining elements, the infinitary ones, do not themselves correspond to particular neighborhoods, but must be regarded as the limits of infinite sets of neighborhoods.

Notice that the finite elements are dense in $|D|$, in the sense that for each x in $|D|$

$$x = \underline{union}\ \{\ \underline{upto}\ X \mid X \epsilon x\ \}.$$

So every element of $|D|$ is uniquely determined by its finite approximations.

Notice also that if $X \subseteq Y$ then $\underline{upto}\ X \supseteq \underline{upto}\ Y$. If $x \subseteq y$, x is less defined than y. In this case we say that x approximates y and often writes this relation as

$$x\ \underline{sub}\ y.$$

Notice that every domain has a least defined element ($\{\Delta\}$), which we call $\underline{|}$ (pronounced "bottom"). If a domain also has elements maximal with respect to the approximation relation they are called total elements. In our first example, for instance, there are two such maximal elements (and the only other element is $\underline{|}$). This domain, in fact, with two fully defined elements, is the one usually taken to model the truth values; in VDM we would call it *Bool*, with elements

$$\{\underline{|}, \underline{true}, \underline{false}\}$$

Moreover, this example easily extends to a larger one, equally useful. Instead of just two tokens, 0 and 1, take all the natural numbers, and let the neighborhoods be the complete set of tokens (of course) and all the singleton subsets. The domain thus produced has, together with $\underline{|}$, a countably infinite number of completely defined (maximal) elements; this domain will be what VDM calls *Nat*. If we rename the elements appropriately, the same domain -- or, if the reader prefers, an isomorphic copy -- will do duty for *Nat0* and *Int* too. (More precisely speaking, a domain comes with some associated primitive operations to give it some extra structure, and it is these which will distinguish *Nat*, *Nat0*, and *Int* -- even though these three domains are isomorphic as neighborhood systems.)

3.3.4 Changing the Token Sets

In our previous two examples the tokens corresponded respectively to the total elements and to all the elements. This is by no means a necessary feature. For example, suppose we are constructing a domain whose elements are intended to correspond to integer sets, and where the approximation ordering is to be the same as the set inclusion ordering. We could take the integers themselves as the tokens and, informally, understand the

presence of token n in a neighborhood to convey the property that the integer n is not a member of the set we are approximating. Thus the finite element corresponding to the neighborhood in Fig. 6 would be {1}. Fig. 7 gives slightly more of the structure of this neighborhood system. It will be seen that although the tokens are countable, there are uncountably many partial elements; but there is only one total element, the complete set of integers.

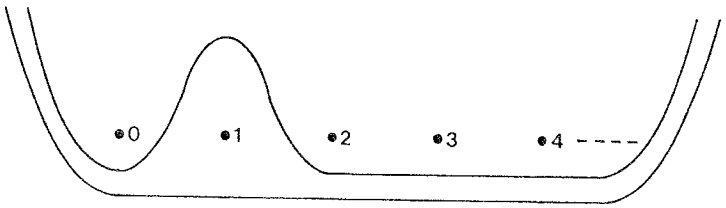

Fig. 6

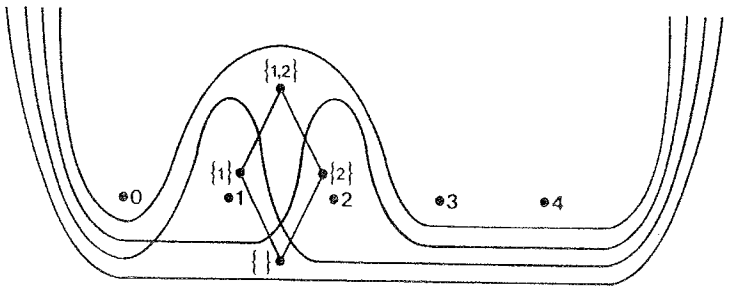

Fig. 7

We remarked earlier that the exact nature of tokens in a neighborhood system was not crucial: the important thing was the neighborhoods themselves. We illustrate this by showing how, given any neighborhood system, we can construct an isomorphic system (that is one with an isomorphic domain of elements structured by the approximation relation and, indeed, an isomorphic system of neighborhoods structured by set inclusion) with different tokens. For example, if D is a neighborhood system, for any $X \epsilon D$ we define

$$[X] = \{\ x \mid X \epsilon x \epsilon |D|\ \}$$

It can be shown that the family of sets like $[X]$ itself forms a neighborhood system and, moreover, that the domain determined by this new system is isomorphic to that of the old one. It will be seen that the tokens of the new system are the elements of the old one, and so are isomorphic with the elements of the new system too.

Alternatively, define

$$\underline{downto}\ X = \{\ Y \mid Y \in D \wedge Y \subseteq X\ \}$$

It can be shown that this family of sets, too, forms a neighborhood system determining a domain isomorphic to the old one. In this case the tokens of the new system are the neighborhoods of the old one, and thus correspond to the finite elements of either domain.

3.3.5 Permissible Functions

We must next consider how to treat functions between domains of elements. Remember that an actual computation about infinitary objects must be conducted solely in terms of their finite approximations. Thus at any stage in the computation of the application of a function to an infinitary argument (another function, perhaps), the argument will be represented only by an approximation, a neighborhood, and we shall require the function to specify similar approximations for the result. That is to say, for neighborhood systems D_0 and D_1 we shall regard a function f between the corresponding domains as a binary relation between the two families of neighborhoods.

It is reasonable to require this relation to satisfy certain properties. In the first place, before the computation starts we have no information about either the argument or the result, which moves us to demand

$$\Delta_0 f \Delta_1$$

Moreover the relation must be consistent, in the sense that

$$Xf Y \text{ and } Xf Y' \text{ imply } Xf(Y \cap Y')$$

Finally, the relation should exhibit a property called monotonicity. That is to say, an improvement in the approximation to an argument may not cancel any definite information we already have about the result. Moreover, whenever any particular neighborhood is specified by the relation as an approximation to the result then any worse approximation must also be speficied. More formally this means that

$$\text{if } X' \subseteq X,\ Xf Y \text{ and } Y \subseteq Y' \text{ then } X' f Y'$$

Definition 4: A relation satisfying all these properties is called an approximable mapping.

It can easily be seen that any approximable mapping between neighborhood systems determines a function between the corresponding domains of elements. For all $x \in |D_0|$ we define

$$f(x) = \{ Y \mid Y \in D \wedge (\exists X \in x)(XfY) \}$$

Conversely, each function on elements determines the original relation on neighborhoods, which is given by

$$XfY \iff Y \in f(\underline{upto}\ X)$$

This two-way correspondence justifies us in using the same letter for the function and the relation. Notice that the function is also monotonic

$x \subseteq y$ always implies $f(x) \subseteq f(y)$

As an example, Fig. 8 tabulates all the approximable maps from *Bool* to *Bool*, listing them both as relations on neighborhoods and as functions on elements.

Tokens: $\Delta = \{0,1\}$
Neighborhoods: Δ, $A = \{0\}$, $B = \{1\}$
Elements: $\underline{|} = \{\Delta\}$, *true* $= \{\Delta, A\}$, *false* $= \{\Delta, B\}$

1.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$	|	| |
2.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfA	|	*true* |
3.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfB	|	*false* |
4.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ BfA	|	| *true*
5.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ BfB	|	| *false*
6.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfA BfA	|	*true* *true*
7.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfA BfB	|	*true* *false*
8.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfB BfA	|	*false* *true*
9.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfB BfB	|	*false* *false*
10.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfA BfA ΔfA	*true*	*true* *true*
11.	$\Delta f \Delta$ $Af\Delta$ $Bf\Delta$ AfB BfB ΔfB	*false*	*false* *false*

Fig. 8 Approximable Maps from *Bool* to *Bool*

3.3.6 Continuity

Although we have been regarding neighborhoods as corresponding to finite approximations of elements, sometimes it is convenient to think of the elements themselves as approximating each other. We therefore need to be able to characterize those sets of elements which are in some sense tending to a limit. The appropriate concept is that of a directed set.

Definition 5: A non-empty set S, partially ordered by a relation $\leq$, is directed if, whenever $x,y \in S$, $x \leq z$ and $y \leq z$ for some $z \in S$.

Notice that z is not necessarily the least upper bound of x and y. A chain of elements

$$x_1, x_2, x_3, \ldots, x_n, \ldots$$

in which $x_n \leq x_{n+1}$ for all n, is a simple example of a directed set.

If S is a directed set of our particular kind of elements (ordered by the relation $\subseteq$), then it is easy to show that

$$\mathit{union}\ \{\ x \mid x \in S\ \}$$

is itself an element. Using more technical jargon, we may say that domains are closed under directed unions.

Now, if $f: D \to D'$ is approximable and $S \subseteq |D|$ is a directed set of elements, we can also easily show that

$$f(\mathit{union}\ \{\ x \mid x \in S\ \}) = \mathit{union}\{\ f(x) \mid x \in S\ \}$$

That is to say, the functions determined by approximable mappings are continuous: they preserve directed unions. This is an important property, giving rise to important techniques for proving properties of our specifications. There is a one-one correspondence between continuous functions on elements and approximable maps between neighborhood systems.

3.3.7 Function Spaces as Domains

We make one further remark about approximable mappings. Suppose D_0 and D_1 are neighborhood systems. Let us construct a further system, in which

the tokens are the approximable mappings between D_0 and D_1 and of which the neighborhoods are the finite non-empty intersections of sets given by

$$[X,Y] = \{ f \mid XfY \}$$

where $X \epsilon D_0$ and $Y \epsilon D_1$. It can be shown that this system is a neighborhood system, and its elements correspond uniquely with the approximable mappings between D_0 and D_1 (it is this that moved us to call the mappings approximable in the first place).

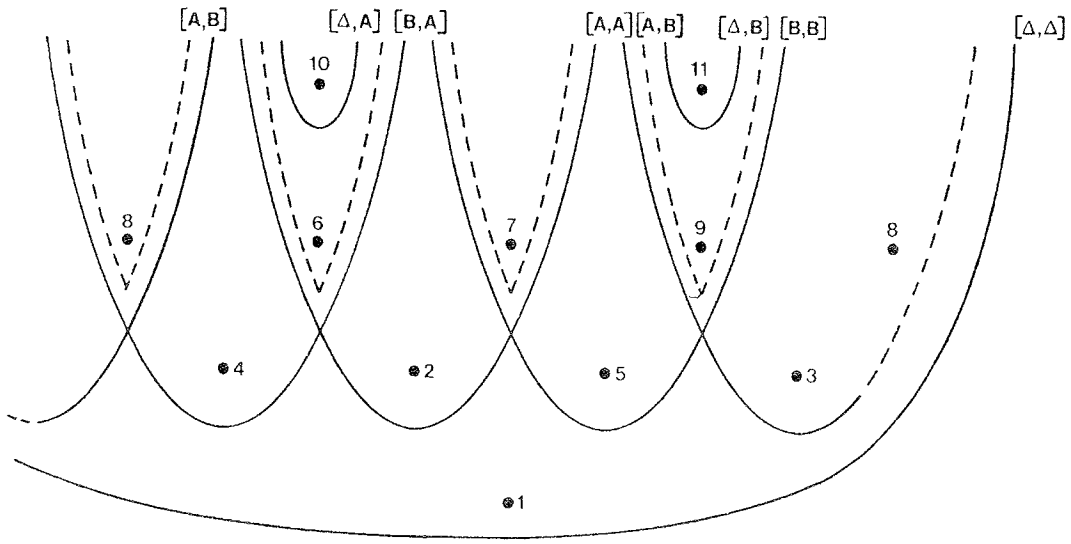

Fig. 9

Fig. 9 shows an example of this construction, for the system corresponding to the approximable mappings from *Bool* to *Bool:* the mappings are denoted by their numbers in the list given in Fig. 8; notice that the neighborhood $[A,B]$ "wraps round, and appears at both edges of the figure. The neighborhoods shown by dashed lines correspond to the "finite intersections" mentioned in the construction.

As a further example, consider the mappings from *Nat* to *Nat*. In this system, assuming that the tokens and elements of *Nat* are given corresponding enumerations, the neighborhood $[\{1\},\{3\}]$ (for example) contains all the functions which map 1 to 3. The neighborhood

$$[\{1\},\{3\}] \cap [\{2\},\{4\}]$$

contains all the functions which map 1 to 3 and 2 to 4. Thus the complete family of neighborhoods in this system has members for all finite maps from *Nat* to *Nat* - that is, those which map only a finite number of integers to anything other than $\underline{|}$. The remaining functions correspond to infinitary elements of the function space. Similarly, if we carry out the construction for

$$(Nat \to Nat) \to (Nat \to Nat),$$

the neighborhoods correspond to finite maps of finite maps: that is to say, in all cases the neighborhoods themselves correspond exactly to what may actually be computed in finite time.

All this indicates that (provided that we confine ourselves to approximable mappings, which we shall henceforth do) function spaces between domains can be considered domains in their own right. We may therefore now begin to consider functions of functions within our framework.

3.4 LEAST FIXED POINTS

In an earlier example of a recursive function definition (3) we noticed that when there was a choice of solutions of the fixed point equations it was the least defined solution (4) we required. We now show that such a least fixed point always exists. That it to say, if $f: D \to D$ is approximable, there is a least x such that $f(x)=x$.

Let us consider how such an x might be approximated. Δ is always a neighborhood of any x, and if X is one neighborhood and XfY, then Y will be another neighborhood. This suggests that

$$x = \{ X \mid X \in D \wedge \Delta f^n X, \text{ for some } n \in Nat \}$$

might be a candidate for our required x. We now show that this guess is correct.

According to our suggestion, $X \in x$ if for some n there exists a sequence

$$X_0, X_1, \ldots, X_n$$

where $X_0=\Delta$ and $X_n=X$, such that $X_i f X_{i+1}$. We must prove first that the x

formed in this way is an element of $|D|$. Now $\Delta \in x$ (consider sequences of length 1); and since f is approximable, $X \in x$ and $X \subseteq Y$ together imply that $Y \in x$. It remains to prove that if $X \in x$ and $Y \in x$ then $(X \cap Y) \in x$. First we note, since $\Delta f \Delta$, the two sequences of neighborhoods relating Δ to X and to Y can be made of equal length by prefixing enough Δs to the shorter one. We also note that if UfV and $U'fV'$ are consistent, then $(U \cap U')f(V \cap V')$. Thus taking intersections element by element of our two sequences we see that $(X \cap Y) \in x$.

Having shown that x is a valid element, we must show that it is the least fixed point of f. We notice that if $X \in x$ and XfY then $Y \in x$, so $f(x) \subseteq x$. Indeed, x is the least such element (since any other must contain Δ and hence all of x too). But, since f is monotonic, $f(f(x)) \subseteq f(x)$; so $f(X)$ is itself another such element, and hence $x \subseteq f(x)$. Combining these two results we have $x = f(x)$ as desired, and we have already seen that x is the least element with this property.

We have thus shown that if we confine ourselves to neighborhood systems, and to functions determined by continuous mappings, then we can assume that recursive function definitions always specify the least solution of the corresponding fixed point equation, secure in the knowledge that such a least solution exists and is unique. So there is a function

$$fix: (|D| \to |D|) \to |D|$$

which maps any function $f: |D| \to |D|$ to its least fixed point. fix may be defined by

$$fix(F) = \underline{union}\{ F^n(\underline{|}) \mid n \in N \}$$

and fix can itself can be shown to be an approximable mapping.

We have thus sketched out how the first of our two problems may be solved. We now turn to the other.

3.5 NEW SYSTEMS FROM OLD

We have already seen one way in which a neighborhood system may be constructed from two others - the space of approximable mappings from one system to another is a neighborhood system in its own right. We now

mention one or two other methods for constructing new neighborhood systems from old.

3.5.1 Product Spaces

Suppose D_0 and D_1 are neighborhood systems over Δ_0 and Δ_1, and suppose Δ_0 and Δ_1 are disjoint (if they were not we could, of course, tag all the tokens in some way to make them so). Then we define

$$D_0 \times D_1 = \{ X \cup Y \mid X \in D_0 \wedge Y \in D_1 \}$$

This can easily be seen to be a neighborhood system itself, over $\Delta_0 \cup \Delta_1$; each neighborhood has a contribution from one neighborhood of D_0 and from one of D_1. It has elements which may be written $\langle x, y \rangle$, where $x \in |D_0|$, and $y \in |D_1|$, defined by

$$\langle x, y \rangle = \{ X \cup Y \mid X \in x \wedge Y \in y \}$$

Thus the elements of this system correspond to pairs of elements drawn from the given systems. The ordering of these elements behaves as expected

$$\langle x, y \rangle \subseteq \langle x', y' \rangle = x \subseteq x' \wedge y \subseteq y' \tag{8}$$

The selector functions may be easily defined too: if $z = \langle x, y \rangle$, then

$$x = \{ X \in |D_0| \mid X \cup \Delta_1 \in z \} = \{ Z \cap \Delta_0 \mid Z \in z \}$$
$$y = \{ Y \in |D_1| \mid Y \cup \Delta_0 \in z \} = \{ Z \cap \Delta_1 \mid Z \in z \}$$

All this indicates that the cartesian product of two domains is a domain too.

3.5.2 Sum Spaces

Once again, let D_0 and D_1 be neighborhood systems over the disjoint sets Δ_0 and Δ_1. Define

$$D_0 + D_1 = D_0 \cup D_1 \cup \{\Delta_0 \cup \Delta_1\}$$

This, too, is a neighborhood system again over $\Delta_0 \cup \Delta_1$; each of its elements, apart from the new minimal one $\{\Delta_0 \cup \Delta_1\}$, corresponds to an ele-

ment of either $|D_0|$ or $|D_1|$. This correspondence is given by the functions

$$in_i: \ |D_i| \to |D_0 + D_1|$$
$$out_i: \ |D_0 + D_1| \to |D_i|$$

where $i \in \{0,1\}$, defined by

$$in_i(x_i) = x_i \cup \{\Delta_0 \cup \Delta_1\}$$
$$out_i(x) = \{X \in x \mid X \in |D_i|\} \cup \{\Delta_i\} \qquad (9)$$

This system, then, corresponds to the disjoint union of the two given domains: the partial ordering is inherited from the two components.

3.5.3 The "Strict" Versions

We can give the following alternative versions of the preceding two constructions.

$$|D_0| \otimes |D_1| = \{\Delta_0 \cup \Delta_1\} \cup \{ X \cup Y \mid X \in D_0 - \{\Delta_0\} \wedge Y \in D_1 - \{\Delta_1\}\}$$
$$|D_0| \oplus |D_1| = \{\Delta_0 \cup \Delta_1\} \cup \{ X \mid X \in D_0 - \{\Delta_0\}\} \cup \{ Y \mid Y \in D_1 - \{\Delta_1\}\}$$

The effect of these variations is to reduce the number of elements in the constructed domains. In the first case, $|D_0 \otimes D_1|$ contains no elements $\langle x,y \rangle$ for which x or y (but not both) is the minimal (least defined) element of the respective component domain: the pairing function defined in (8) takes all such elements to the minimal element of the new domain ($\underline{|}$, which is also $\langle \underline{|}, \underline{|} \rangle$). In the second case, $|D_0 \oplus D_1|$ contains no separate elements corresponding to the minimal elements of the component domains: in_i (for each i) maps each to the minimal element of the sum domain. This disjoint construction, with the minimal elements identified in this way, is sometimes known as the coalesced sum, as distinct from the the separated sum given earlier. Which version of the sum or product domain construction is required depends on the particular circumstances (though VDM does not provide separate notation for both). We shall have more to say about this when we come to apply this work to the VDM notation more specifically.

3.6 RECURSIVE DOMAIN EQUATIONS

We have now discussed the three principal ways of combining given domains

into bigger structures - by forming product, sum or function spaces. (There are, of course, other constructions, but they are not too relevant to VDM.) So we now know how to interpret an equation such as

$$Val = Bool + Int + (Int \times Int)$$

provided *Bool* and *Int* have been defined (they were discussed in examples above). We must now consider equations in which a domain is described circularly, as in the following example

$$Tree = Int \oplus (Tree \otimes Tree)$$

From this equation we may infer, for a start, that there is an element of *Tree* corresponding to each element of *Int*, and to every pair of elements of *Tree*. This structure (together with the actual $Int \rightarrow Tree$ and $(Tree \times Tree) \rightarrow Tree$ functions specifying the correspondence) would amount to an algebra of trees. But we shall require more. An algebra might associate (for example) two different pairs of trees ($\langle A,B \rangle$ and $\langle B,A \rangle$, say) with the same element of *Tree*. We shall forbid such "confusion". The fact that we have written a domain equation is meant to imply that the two sides are to be isomorphic: two elements of the domain are to be identified only if they arise from the same structure of components. Even this is not enough, however, to specify a unique solution of the equation. The domain *Tree* (for example) might include extra elements which, though their presence does not invalidate the domain equation, are not essential for a solution. We therefore forbid such "junk" elements, too. Elements are to exist in the domain only if their existence is required (not merely permitted) by the defining equation.

An example of unnecessary elements for our particular equation is afforded by infinite trees, such as

$$\langle 1,\langle 1,\langle 1,\ldots\rangle\rangle\rangle$$

The presence of such trees would not invalidate our equation, but they are not required by it; so they are not present in the particular solution we have in mind. Notice, though, that if we had used the alternative (uncircled) versions of the domain construction operators some infinite trees (those which are the limits of their finite approximations) would have been part of the solution too. For example, all the elements of the sequence

$\underline{|}$, $\langle 1, \underline{|} \rangle$, $\langle 1, \langle 1, \underline{|} \rangle \rangle$, $\langle 1, \langle 1, \langle 1, \underline{|} \rangle \rangle \rangle$, ...

will be present, and therefore (since, as we have seen in section 3.3.6, domains are closed under directed unions) their least upper bound will also be present, and this is the infinite tree we have mentioned - otherwise our solution would not be a domain at all. The use of the strict operators (in particular ⊗), however, will ensure that any attempt to produce a tree containing $\underline{|}$ will in fact produce $\underline{|}$ itself; thus no (non-trivial) finite approximations of infinite trees are in the domain *Tree*, and neither are the limitpoints. Another way to see the difference between the two versions of *Tree* may be to consider what is the least solution of the equation

$$x = \langle 1, x \rangle$$

in the two domains - it will be $\underline{|}$ in the strict version, but the infinite tree in the other. (Notice that the version for which infinite trees are present by necessity may nevertheless have solutions which include other "junk" elements -- an example would be the domain which also included infinite trees of Boolean atoms.)

The particular kind of solution that interests us ("no confusion, no junk", in Burstall's phrase) corresponds to the initial algebra and is, in the sense which we have outlined, the unique minimal solution of the equation. We now have the obligation of showing that an initial solution exists for any such equation. One way of doing this is, firstly, to define the implicit partial ordering on domains.

Definition 6: *D* is subsystem of *E*, written "*D sub E*", (and the corresponding domains are in the subdomain relationship) if *D* and *E* are both neighborhood systems over the same set of tokens and

1. $D \subseteq E$;
2. whenever $X, Y \in D$ and $X \cap Y \in E$, then $X \cap Y \in D$

That is to say, *D* is a "smaller" family of neighborhoods, but neighborhoods in it are consistent whenever they are consistent in *E*.

Then we consider

$$T(X) = Int \oplus (X \otimes X)$$

as defining a "function", T, from the class of domains (into itself), and we define what it means for such a "function" to be monotonic and continuous. Then if we have a token set Δ such that

$$\{\Delta\} \underline{sub}\ T(\{\Delta\})$$

we can show that the required initial solution of

$$D = T(D)$$

is given by

$$D = \underline{union}\ T^n(\{\Delta\})$$

Notice that $\{\Delta\}$ is the minimal neighborhood system over Δ, and if it is a subsystem of $T(\{\Delta\})$ then the whole sequence of systems will be over the same set Δ. (Of course this is no more than the barest of possible outlines - strictly speaking, for example, T is not only a function on the class of domains but has to do the right thing to the approximable mappings on those domains too. The neatest way to say it properly is in the notation of category theory, making T a functor, and the interested reader will find it all in [Scott 81a].)

3.7 AN ALTERNATIVE APPROACH TO DOMAIN EQUATIONS

In a sense we have now dealt with all the problems outlined in section 3.2. We have shown that, under constraints which hold for the cases with which we are concerned, solutions exist for our circular definitions both of functions and of domains themselves, and we have discussed how it is always the minimal solution that interests us in particular. We choose, however, to approach the matter of domain equations from another direction. The point is that the methodology for which we are seeking to provide the foundations is to be used for specifying computer languages and systems; so, if possible, we should try to keep track of whether what we are specifying is computable. We must therefore extend our treatment to take computability into account, and we shall find that the alternative approach will allow us to do so more easily.

3.7.1 Computability of Domains and Mappings

We recall that the basis of this model is that the neighborhoods represent the finite approximations to our finitary or infinitary values: so it is with neighborhoods that we actually compute. In order that these computations with neighborhoods be possible, we require the neighborhood system to be effectively presented - that is to say, we require neighborhoods inside the machine (or on paper) in such a way that the necessary calculations can be carried out. This in turn means that there must be no more than countably many neighborhoods, so that we may think of them as indexed by the natural numbers (that is, we may think of a typical neighborhood as X_n for some natural number n); More than this is needed, however: for we must be able to tell which neighborhoods are which, and how they relate to each other, in an effective way. The formal requirements are spelled out in the following definition.

Definition 7: A neighborhood system D has an effective presentation (or "D is effectively presented") if

$$D = \{ X_n \mid n \in N \},$$

where the two propositions

$$(\exists k \in N)(X_k \subseteq X_n \wedge X_k \subseteq X_n), \text{ and } \quad X_n \cap X_m = X_k$$

are recursively decidable (in m,n and k,m,n respectively).

Elements, and particularly infinitary elements, are thought of as the limits of their finite approximations. So all we have a right to expect, when computing a particular element, is that any particular approximation to that element will be produced sooner or later. That is to say

Definition 8: An element x of an effectively presented domain is said to be computable if the set

$$\{ n \in N \mid X_n \in x \}$$

is recursively enumerable.

In a similar way we define what it means for an approximable mapping to be computable.

Definition 9: An approximable mapping $f: X \to Y$, where X and Y are effectively presented, is computable if the relation

$$X_n f Y_m$$

is recursively enumerable in m and n.

The reader unfamiliar with recursive function theory need not be too concerned to study the details of these definitions. Suffice it to say that they express our requirements using the standard apparatus, and that when they overlap with the standard theory (for example, when they are applied to the domains of integers and integer functions) they are precisely compatible. We may now go back over our previous working, and check that we can find effective presentations for the domains we have introduced, such that any "primitive operations" specified for any particular domains - or any operations defined for families of domains - are computable in terms of that presentation. In this summary the only example we have explicitly mentioned is fix (this is, incidentally, the only fixed point operator which is in general computable). We should check that our domain constructors ($\times$, + and so on) produce domains which are effectively presented whenever the constituents are (that is to say, we can work out algorithms for the necessary calculations with the new neighborhoods given those for the constituent domains). Note, too, that a function is computable, considered as an approximable mapping, just when it is computable considered as an element of the function space domain.

We now give a method for finding solutions to domain equations that better allows us to keep track of this notion of computability.

3.7.2 Retracts

Our new programme for finding solutions of domain equations involves exploiting the subdomain relation *sub*. Our plan is to find the solution of any domain equation as a subdomain of one particular "universal" domain which contains them all. One convenient way to characterise a particular subdomain of a given domain is as the range of some function; an even better plan is to confine our attention, if we can, to functions which are idempotent (that is to say, which are the identity function on their range, so that their ranges and fixed-point sets coincide). Such a function is called a retraction, and its range set a retract, of the given domain.

Definition 10: A retraction of a neighborhood system E is an approximable map $a: E \to E$ such that

$$a \circ a = a$$

Now to relate retractions and their retracts with subdomains, we note that if D sub E, then there exists a pair of approximable maps relating the two domains, namely $i: D \to E$ and $j: E \to D$ defined as follows

$$i(x) = \{ Y \mid Y \in E \wedge (\exists X \in x)(X \subseteq Y) \}$$
$$j(y) = y \cap D$$

These two maps are known as a projection pair; note that

$$j \circ i = I_D \text{ and } i \circ j \subseteq I_D$$

If we look more closely at this last-mentioned $i \circ j$, which may alternatively be defined as the approximable mapping a given by the relation

$$X \; a \; Z \iff (\exists Y \in D)(X \subseteq Y \subseteq Z),$$

we see that it is a retraction, since

$$a \circ a = i \circ j \circ i \circ j = i \circ j = a,$$

and, moreover, that $|D|$ is isomorphic to the fixed-point set of a. To show this last point, we note, on the one hand, that for any $x \in |D|$, $i(x)$ is an element of a's fixed-point set, since

$$a(i(x)) = i \circ j \circ i(x) = i(x);$$

and, on the other hand, that for any fixed point y of a there is an element of $|D|$, namely $j(y)$, such that

$$i(j(y)) = y$$

So i sets up a 1-1 correspondence between the fixed-point set of a and the domain $|D|$ which (since i and j are both monotonic) is an isomorphism under the $\subseteq$ ordering.

Notice that not every retraction corresponds to a subdomain of the given

domain in this kind of way. We call a retraction a of E a projection if

$$a \subseteq I_E$$

and we call it a finitary projection if its fixed-point set is indeed isomorphic to a subdomain. The finitary projections of E are just the functions a which satisfy

$$a(x) = \{ Y \mid Y \in E \wedge (\exists X \in x)(X \; a \; X \wedge X \subseteq Y) \}$$

3.7.3 A Universal Domain

Our final step in this development is to produce a domain which is "universal" in the sense that any required domain may be found as one of its subdomains. There are many possible candidates - even if we confine ourselves to domains which are effectively given - a convenient one uses as its tokens the rational numbers in the half-open interval $[0,1)$, where

$$[r,s) = \{q \in Q \mid r \leq q < s\}$$

We define the system U over $[0,1)$ to have as neighborhoods all non-empty finite unions of intervals

$$[r,s) \text{ where } 0 \leq r < s \leq 1$$

This can easily be seen to be effectively given. This system U is universal in that every countable neighborhood system D is a subsystem of U and, moreover, if D is effectively given the relevant projection pair (and hence the associated retraction) is computable.

Next we find isomorphic versions of our favourite primitive domains as subdomains of U. *Bool*, for example, may be found as the neighborhood system

$$\{[0,1),[0,1/2),[1/2,1)\}$$

giving rise to the domain

$$\{\{[0,1)\},\{[0,1),[0,1/2)\},\{[0,1),[1/2,1)\}\}$$

The reader might care to define a suitable retraction function for this

domain (remember that when applied to any element of U it must produce an element corresponding to an element of $Bool$, and those particular elements must be mapped to themselves), and perhaps also for a suitable version of the domain of integers as a subdomain of U.

Finally we must cover the constructors for compound domains. First we note that (because they are effectively given) $U \times U$, $U + U$, $U \to U$ are themselves sub-domains of U, and so we can assume the existence of projection pair functions

$$i_{\times}: U \times U \to U \text{ and } j_{\times}: U \to U \times U$$

and similar functions for the other forms of construction. (When we came actually to design a suitable pair of functions $i_{\times}$ and $j_{\times}$ we would have to take care to make the tokens corresponding to the two subdomains disjoint -- perhaps by mapping them to $[0,1/2)$ and $[1/2,1)$ respectively). Now for any $a,b \in U \to U$, define

$$a \times b = i_{\times} \circ (\lambda x.\langle a(x[0]), b(x[1])\rangle) \circ j_{\times}$$

where $x[0]$ and $x[1]$ are the two components of x whenever x is a pair, and $\underline{|}$ otherwise. Similarly, we may define

$$a \to b = i_{\to} \circ (\lambda f.b \circ f \circ a) \circ j_{\to}$$

and so on for the other constructions. If a and b are finitary projections then so are $a \times b$, $a \to b$ etc. and their fixed-point sets are isomorphic to the domains obtained by applying the corresponding domain constructor to the domains characterised by a and b. This programme may be continued: an important step is to show that if f is a function such that $f(a)$ is a finitary projection whenever a is, then $fix(f)$ is also a finitary projection.

Thus, instead of working with the domains themselves, we may work instead with their finitary projections. Since, if a and b are finitary projections of U

$$D_a \underline{sub}\ D_b \quad \underline{just}\ \underline{when} \quad a \underline{\leq} b,$$

this new view is isomorphic with the old. So we may now think of a recursive domain equation as defining the domain given by the correspond-

ing recursively defined finitary projection: both equations specify the minimal solution (just as we understood circular domain equations to do in the earlier approach), but now we know when the resulting domains are effectively given and accordingly appropriate vehicles for specifying a computation.

3.8 APPLICATION TO VDM

We must finally discuss how these notions map onto the VDM matalanguage that is the subject of this book. This is partly a notational change - our notation up to now has been very similar to Scott's. For example, VDM uses domain names such as *State*, unadorned with bars, to denote families of elements which we have denoted, up to now, by $|State|$.

3.8.1 Primitive Domains

The primitive domains of VDM correspond to systems we have already discussed. We should remember, however, that these systems are structures consisting of a neighborhood system and a number of primitive operations, in terms of which we could define all other operations involving elements of the domain. For example, the non-negative integers (the domain VDM calls *Nat0*) could be defined as the structure

$$\langle N, 0, pred, succ, zero \rangle$$

where N is the countably infinite system introduced in section 3.3.3. Other numeric domains (such as *Int* and *Nat*) would correspond to similar, but different structures. The truth-value domain, *Bool*, would have its own family of operators including, for any other domain D, the conditional combinator

$$Cond: Bool \times D \times D \rightarrow D.$$

The other primitive domains (domains of Quotations (Characters), or of explicitly enumerated elements - such as the domain {CLUBS, HEARTS, SPADES, DIAMONDS}, for example) have their own families of operators too: these often consist merely of an equality test, sometimes together with some ordering relation.

For all these structures we must check that we can first define suitable

enumerations of their neighborhoods, and then define the primitive operators as computable functions in terms of these enumerations. This presents no problem for the domains we have discussed.

3.8.2 Compound Domains

Our job for the compound domains is very similar - we must provide enumerations for the neighborhoods (in terms of the enumerations for the component domains), and definitions for a sufficient set of primitive operators. We should also clarify what properties (if any) we need to assume about the component domains in our construction.

Many of the domain constructions require little discussion. The construction for product spaces, for example, has been sufficiently described, and the VDM notation is the same as used in this chapter.

3.8.3 Sum Domains

For sum spaces the VDM notation is $A \mid B$. In many cases this is best regarded as equivalent to our $A \oplus B$; for example, in the specifications of syntactic domains (where, of course, the sum domain notation is used to give alternatives for nodes in the parse tree) we do well to rule out the infinite parse trees that would be elements of our domain if we used the non-strict versions of the sum and product operators. Occasionally, though, the other version is appropriate. For example, consider a semantic value domain defined to be

$$Val = Int \mid Bool \mid Proc$$

Here *Proc* is supposed to be a domain of procedure values, presumably functions from parameter lists to some appropriate space; so its minimal element will be the function mapping all parameter lists to $\underline{\perp}$. This is indeed the least-defined procedure, but it must not be confused with the minimal element of *Val* itself. $\underline{\perp}_{Val}$ is such that any attempt to evaluate it will fail to terminate - in our terminology, no neighborhood will be produced in the enumeration of neighborhoods except for the trivial neighborhood (Δ) conveying no information beyond the fact that the value is an element of *Val*. The minimal procedure, however, is at least a procedure, and as such might be passed around as a parameter to other procedures or whatever; it is only when it is applied to a parameter list that a non-terminating computation might be expected to result.

(In fact this point does not arise in the language given in Chapter 4 - the only place where it might is in the definition of the domain *Den* by

$$Den = Loc \mid Labden \mid Procden$$

but in this case since $\underline{\perp}_{Procden}$ is the only element that can ever arise there is no real need to distinguish it from any other.)

The best rule of thumb in deciding whether the strict or the non-strict constructions are required is probably to use the strict versions whenever the component domains are all flat (that is, they contain only maximal elements and $\underline{\perp}$): this will result in a new domain which is also flat, thus avoiding extra complication which is almost always unnecessary. When a component domain has a richer structure, however, or alternatively when the construction is part of a circular definition of a domain that is to include infinitary elements, the non-strict versions are usually the appropriate ones to use. Note, however, that this is merely a rule of thumb, and often more careful analysis is required. Even in the example we discussed in section 3.6, the various different choices for the operators result in domains which are quite different, and would be appropriate for list systems with quite different semantic properties. Donahue and Cartwright (in [Donahue 82a]) discuss just this taxonomy for a comparison of various definitions of "lazy evaluation".

3.8.4 Abstract Syntax

We must now discuss the difference between the use of "=" and of "::" as the defining operator in a domain equation, since either may be used, particularly when a product domain is being defined. In fact there is nothing abnormal about the "=" operator; in particular, if two domains are defined using this operator with the same right hand sides the domains will be identical. The "::" operator, on the other hand, always defines a domain together with a family of named constructor and selector functions, and names for the test functions which are implicitly defined when the domain is used in sum domain constructions with others. Thus the difference between the two operators is basically one of associated naming conventions.

3.8.5 Functions

The VDM notation for function spaces corresponds closely to ours, though

an additional distinction may be made, between a domain of total functions and one containing partial functions too. Since this is well known to be an uncomputable distinction, we regard it as making no difference to the domain, which will be the same in either case; it is merely a matter of more or less informal comment, indicating that those writers who use both symbols in their definitions are, when they use the "total" version, perhaps prepared to prove that the functions they are defining are total. (The vagueness of this remark is intentional, since in practice some authors tend to use the plain arrow for the unrestricted class.)

3.8.6 Sets

The notation provided for specifying set domains in VDM is so general that it is possible to define domains which are not effectively presented and for which some of the operators provided in VDM are uncomputable. For example, unless it is possible to tell effectively whether two elements of a set are equal it will not be possible to compute the set's cardinality. We therefore allow as elements of set domains only finite sets, of elements of some flat domain (but excluding $\underline{|}$). Since the cardinality function *card* must, as usual, be monotonic, the domain of sets must be ordered in such a way that sets of different cardinalities are incomparable: the set domain, in fact, is a flat domain too. If the domain of members is countably infinite (no effectively given flat domain can be uncountable), the set domain will be yet another domain isomorphic to the domain of integers, as can be seen by considering the standard representation of sets as bit patterns, and regarding them as binary numbers. This representation technique immediately suggests a strategy for formulating the details of the neighborhood system and the primitive operations, and for defining a suitable projection pair, to allow such set domains to be viewed as subdomains of our universal domain U. When this is worked out, it will be seen that the construction makes use of more properties of the component domain than any previous one (it will, for example, rely on its flatness).

It should be remarked that other kinds of domains of sets are also sometimes used in denotational definitions (though not, as yet, in the VDM notation). One example is the domain of sets of possible results of nondeterministic computations. The theory of these "powerdomains" is quite different from that for the much simpler domains we are considering here - for example, such domains need not necessarily be effectively given (the computer, after all, produces only <u>one</u> of the set of possible

results). For a discussion of the various forms of powerdomain see [Plotkin 76a] or [Smyth 76a]. or (for the kind which does fall inside the framework of neighborhood systems) [Scott 81a].

3.8.7 Lists

One possible way of constructing a domain of A-lists, where A is some domain, is as the initial solution to the equation

$$L = L \oplus (A \otimes L).$$

It is straightforward to define the appropriate set of primitive operators in terms of this domain.

3.8.8 Maps

Maps in VDM are finite functions on flat domains. Moreover, given a map, it is possible to compute the element of a set domain containing the elements for which the function is defined, implying that maps defined on sets of different cardinality are incomparable.

One possible way to provide a construction for map domains is to proceed by representing a map as a pair, consisting of the set on which it is defined together with an approximable mapping of the usual kind. All the VDM operators involving maps are then straightforward, except possibly for those involving the range. For example. if m is a map from A to B, $\underline{rng}\ m$ is allowable only if a domain may be defined for elements of B (see above); even then, $\underline{rng}\ m$ will be computable only if m happens to be a total map on its domain.

Maps in VDM, however, are most frequently (though certainly not always) used in definitins where the members of their range sets are known to be elements (excluding $\underline{|}$) of some flat domain. Then the map domain is flat too, and none of the problems just described arises. Thus, for example, chapter 4 section 8 contains the definition

$$STORE = SCALARLOC \overset{m}{\to} [SCALARVALUE]$$

where

$$SCALARVALUE = Bool \mid Int$$

and *SCALARLOC* is also a flat domain. We see that *STORE*, too, is flat and, and, therefore, so is the domain of states in that definition (since *State* is a product of *Store* together with other flat domains).

o o o

The discussion of this section may seem to be calling for some fairly conventional programming, albeit in terms of neighborhoods. It should be remembered that the object of such exercises is to check that the domains being considered can be effectively presented, and that they can be provided with an adequate set of computable primitive operators. This does not in any way prejudice any decision about how an actuial implementation might represent the elements of any domain. It does, however, provide us with quarantees on the following points which would otherwise, if necessary, have to be proved explicitly.

1. The domains may be used in recursive definitions of other effectively given domains;

2. the domains may be used as the basis for the specification of functions which we may reasonably expect to implement on a computer.