APPLICATIONS OF CONTINUOUS LATTICES TO λ-CALCULUS AND DENOTATIONAL SEMANTICS

BY PAUL TAYLOR,

TRINITY COLLEGE, CAMBRIDGE.

MAY 1983

An essay proposed for Part III of the Mathematical Tripos.

Contents

I	Scott's argument	1.
II	Lambda Calculus and the $P\omega$ model	6.
III	The Retract Calculus of Data Types	12.
IV	Self-acting Monoids	18.
V	The Semantics of Programming Languages	25.
VI	Powerdomains and Nondeterminism	33.
	Footnotes and Concluding Remarks	39.
	Bibliography	42.
	Acknowledgements	45.

CHAPTER I SCOTT'S ARGUMENT.

The theme of this essay is Dana Scott's work on providing a mathematical semantics for programming languages and constructing a model for Church's lambda calculus, leading to the theory of continuous lattices.

The guiding question is what it means to perform an *infinite* calculation on a *finite* computer. For example, to say that $\pi = 3.14159$ is plainly false[1], but if we build a machine which prints out successive decimal digits of π and which is such that we can guarantee that each digit will eventually be printed, we can say that it *is calculating* π, which is in some sense the *union* of the partial results obtained after all finite times, each of which we can say is an *approximation* to π. More precisely, 3.14159... means the approximation $[3.14159, 3.1416]$ and $\pi = [3,4] \cap [3.1, 3.2] \cap [3.14, 3.15] \cap [3.141, 3.142] \cap \ldots$

Thus Scott [1970] postulated that a *data-type*, D (in this case the reals, $D_{\mathbb{R}}$) should be considered to be a set

(i) partially ordered[2] by $\leq$ (*approximation*)

(ii) with a least element $\bot$ (*bottom: no information*)

(iii) closed under directed sups.

Such a device is called a <u>complete partial order</u> (cpo[3]).

In the example above, $D_{\mathbb{R}}$ is the collection of closed sets in $\mathbb{R}$, partially ordered by *reverse* inclusion, so $D_{\mathbb{R}} \cong \Omega(\mathbb{R})$, the usual topology on $\mathbb{R}$. Then $\bot = \mathbb{R}$, $\top = \emptyset$ (top) and sup is intersection. Another example we shall consider is $P(X)$, the set of subsets of a set X partially ordered by inclusion. Any finite subset of X is <u>compact</u>: if it is covered by a directed set in $P(X)$ then it is covered by some element of the set; only $\bot \in D_{\mathbb{R}}$ has this property (trivially). $P(X)$ is said to be <u>algebraic</u> since every subset of X is the directed sup (union) of the compact (finite) sets below (inside) it.

We shall return to $P(X)$ for $X = \omega$ (the natural numbers) in the next chapter, and to more general algebraic lattices in chapter III.

The order, $x \le y$, is to be interpreted as *x has less information than y*, so $\bot$ means *undefined*. $\top$ means, if anything, not *perfect information* but *overdefined* or possibly *inconsistent*; however we shall see in chapters V & VI that it is in fact more natural to exclude it altogether. The directed sup is *the ultimate result of a process*; in chapter VI we shall discuss the binary sup[4] as the result of *cooperation between two processes*.

What can we say of the maps between our domains, D? Well first, they are monotone. For if $x \le y$ are approximations, in calculating $f(y)$ we may expect that to know y is at least as good as to know x, so $f(y)$ is at least as good as $f(x)$, ie $f(x) \le f(y)$. However f must also preserve directed sups (which, incidentally, implies monotonicity). For to get a *finite* amount of information about $f(\bigvee S)$ we should need only a *finite* amount about $\bigvee S$, and *all* of the information about $f(\bigvee S)$ should be given by *all* the $f(x) \le f(\bigvee S)$, so $f(\bigvee S) = \bigvee f(S)$ for directed S. If f satisfies this we call it [Scott-] continuous[5]; if also $f(\bot) = \bot$ we say f is strict.[6]

We have still not caught the whole of the notion of *approximability*: we still need some notion of *finitely calculable* which is such that every element is the directed sup of the finitely calculable things inside it. [A set which generates the lattice in this sense and is closed under binary inf is called a basis]. It appears that to make such a requirement *absolutely*, ie that the lattice is algebraic, is too strong[7], so we formulate such a condition *relatively*.

If we know *eventually* that π is (in) [3.14159, 3.1416] then certainly some *finite* calculation will tell us it's in [3.141, 3.142], but not necessarily [3.1415, 3.1416]. Both are *approximations*, but the former is more strongly so: it is an *essential chunk* [Stoy 1977, p.107] of the information. Formally, we define $x \ll y$ [read, *x is way below* or *is relatively compact in*] if whenever $y \le \bigvee S$ for some directed set S, then already $x \le s$ for some $s \in S$. Then $x \ll x$ iff x is compact. In topology, we have $U \ll V$ for two open sets iff $\overline{U} \subseteq V$, so U *misses* V *even at limits*.

A continuous lattice is a complete lattice (a poset with all sups and infs[8]) such that for all x,

$$x = \bigvee \{ u \ll x \}$$

For further information see the Compendium[9] [≡ Gierz et al. 1980].

Now since $x' \leq x \ll y \leq y'$ implies $x' \ll y'$, every algebraic lattice is continuous, but $D_{\mathbb{R}}$ shows that the converse is false. In fact the topology of a space satisfying some suitable separation axiom [such as regularity, Compendium I 1.8, or sobriety, ibid. V 5.6] is a continuous lattice iff the space is locally compact.[10]

Scott[11] [1980, 1982] has given two further formulations of the notion of data types as lattices, which are called, respectively, neighbourhood systems and information systems. In the former he introduces an *approximable function* as a relation between elements (neighbourhoods) in the source and target as *if you know x, then you know at least y about f(x)*, which he shows to be equivalent to Scott-continuity.

Complete partial orders have a topology, the <u>Scott topology</u>, with respect to which a map is continuous iff it's Scott-continuous in the earlier sense. A set is <u>closed</u> (and its complement is <u>open</u>) if it's closed under approximation and directed sups, ie $x \leq y \in A$ implies $x \in A$, and $S \subseteq A$ directed implies $\bigvee S \in A$. [Compendium II 2.1].

Given two continuous lattices, L, L′, we may consider the collection of maps between them subject to the pointwise order. This forms another continuous lattice, which is denoted $[L \to L']$ [Compendium II 2.16; also *infra* p. 21]. It is important that this is essentially no bigger than L and L′ [Compendium III 4.2] because then it is possible that $L \cong [L \to L]$ which is not possible for sets because of Cantor's theorem.[12]

This enables us to find a model for the *untyped* lambda calculus. Most languages (even FORTRAN) allow functions to be passed as parameters to procedures and some (such as LISP) allow them to be returned as results. Thus unless we have rigid type restrictions, we have the problem of *unrestricted self-application*.

Since we are dealing with the semantics of *programming* languages, procedures as well as data types must be considered. Scott [1970a] also applies the foregoing lattice theory to this. The method pioneered by him and the late Christopher Strachey also deals with side-effects, jumps and assignments and provides tools for *program-proving*; it will be discussed in Chapter V.

Scott describes an algebra of *straight-through* (ie without loops) flowcharts, dealing with composition and conditional execution. Approximation is then defined as *missing out* parts of the flowchart. We have a lattice of flowcharts with at most n components (we have to construct products and unions of lattices) which form a nested sequence (consider the analogous construction of a tensor algebra over a vectorspace)[13] which has a limit. This contains both our *finite* flowcharts and certain *infinite* ones which are directed sups of finite ones.

What do these *infinite* elements of the lattice mean? Consider the action of a **while**[14] loop (which we did not include in the earlier specification). In any *actual* (terminating) calculation, the loop is traversed a finite number of times and then exited, and so in this case the procedure is equivalent to a (repetitive) straight-through procedure. Moreover, under the approximation ordering, we have an increasing sequence, of which we may say the **while** loop is the limit[15] (sup). Thus

$$\ulcorner \textbf{while } E \textbf{ do } \Gamma_0 \urcorner = \Gamma = \ulcorner \textbf{if } E \textbf{ then } (\Gamma_0 ; \Gamma) \urcorner$$

Here we have an example of a recursively-defined procedure, analogous to the (inevitable) example of the recursively-defined function

$$\ulcorner \mathit{fact}(n) \urcorner = \ulcorner \textbf{if } n=0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}(n-1) \urcorner$$

so the question arises whether such equations necessarily have solutions.

Fortunately (for we shall be using such equations frequently and without comment) the answer is yes, because of Tarski's fixed-point theorem. If $f: D \to D$ is a continuous map on a cpo, then f has a least fixed point, given by $\bigvee \{ f^n(\bot) : n \in \omega \}$, where f^n denotes repeated application. Moreover, if D is a complete lattice, the fixed-points of f also form a complete lattice [Compendium O 2.3]. A generalisation of this to categories will be given in chapter IV.

Finally, a note on cardinality. For applications to computer science it is natural to assume all our lattices to be countably based, and countability is often incorporated into many definitions. Thus $x \ll_\omega y$ is a

weaker form of $x \ll y$ which mentions only countable directed sups, and ω-algebraic and ω-continuous are weaker conditions on lattices. We shall not employ this convention, but certain things will occasionally be assumed countable; however this is rarely crucial and most of the theory goes through identically with any fixed cardinal in place of $\aleph_0$. With countability one does not generally need the Axiom of Choice, but in so far as some things are allowed to be uncountable it will be assumed throughout.[16]

Countable directed sups are easily shown to be equivalent to sups of ω-chains (ie with terms indexed by natural numbers): frequently directed sups (or, in categories, filtered colimits) are thus referred to as unions of chains. Given (co)limits of all λ-chains for $\lambda \leq \kappa$ we also have those for directed systems of cardinality at most $\kappa \geq \aleph_0$. Indeed if the category is small ($<\kappa$) we then have all such (co)limits.

CHAPTER II LAMBDA CALCULUS AND THE $P\omega$ MODEL

Lambda calculus deals with functions under *application* rather than *composition* as in Category Theory. The latter suggests a *strong typing* discipline, in which we begin with some basic type(s) and build up sums, products and function spaces recursively as *distinct* types, so that (in particular) a function can never be applied to itself. Whilst this is a laudable *programming practice*, it tends not to be what is actually *permitted*, and against the advantage that every expression has a (unique) normal form [Bar. 561] is set the (equivalent) disadvantage that there is no fixed--point operator, Y [*ibid.* and *infra*].

For the purposes of symbolic logic and compiler--writing, some care must be taken over the syntax of λ-expressions, in particular over bound and free variables [Bar. 22-35 & 575-9; Stoy 57-67] but such pedantry is inappropriate here. The important notion is that of abstraction: thus if $f(x)$ is some *expression* in the (free) variable x, we have the *function* $x \mapsto f(x)$, written $\lambda x. f(x)$. Since the former notation is perfectly adequate for this function, the "λ" will be interpreted here as a device which takes the *function* $x \mapsto f(x)$ and gives it a *name* $\lambda(x \mapsto f(x))$ in our original universe, since we intend to have the *functions* $f: D \to D$ as *elements* $\lambda f \in D$ in the domain D. Thus

$$\lambda : [D \to D] \hookrightarrow D \qquad (\xi)$$

where $[D \to D]$ is the function space as constructed in the previous chapter; we also have $D \times D$ with the componentwise order, so that $[D \times D' \to D''] \cong [D \to [D' \to D'']]$ [Compendium II 2.10].

In order to force injectivity, we shall provide λ with a post-inverse, $\bar{\varepsilon}$, corresponding under the adjunction just mentioned (which is called currying) to application, ε, where $\bar{\varepsilon}: D \twoheadrightarrow [D \to D]$ and $\varepsilon: D \times D \to D$. Then

$$\varepsilon(\lambda(f), x) = f(x) \qquad (\beta)$$
$$(\lambda x. M)N = M[x := N]$$

both represent the axiom of β-reduction, where $M[x := N]$

denotes M with each free occurence of x replaced by (N). [Bar. 23; Stoy 65]. If $(D, \lambda, \varepsilon)$ satisfies (β) we shall call it a <u>$\lambda\beta$-model</u>; we shall discuss the stronger condition that $[D \to D] \cong D$, called (η), in chapter IV.

$$\lambda x.\ Mx = M \qquad (\eta)$$

Formally, <u>λ-expressions</u> are built up inductively from (countably many) variables $x, y, z, \ldots$ by

(i) $x \in \Lambda$
(ii) $M \in \Lambda \Rightarrow (\lambda x. M) \in \Lambda$
(iii) $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$

The set of such expressions, Λ, is clearly countable; a λ-expression which has no free variables will be called a <u>combinator</u>[3] and the set of them denoted Λ_0. We have a rule, of purely syntactic significance, that $(\lambda x. M) \equiv (\lambda y. M[x := y])$, which is called ($\alpha$); this is of course subject to the obvious self-discipline over variables. The substantive rules (β) and (η) have already been mentioned.

In the implementation of most programming languages, each of the arguments of a function is evaluated before any attempt is made to apply the function, even if some or all of the arguments are unused; this is called <u>applicative order</u> evaluation. The alternative is <u>normal order</u> (or <u>lazy</u>) evaluation, which reduces an expression to its (unique) <u>normal</u> (ie β-irreducible) <u>form</u> if it has one, by applying (β) in the leftmost available place [Bar. 322-5; Stoy 67-8] Here a calculation is performed only when explicitly required, so that a program to find a solution of a problem may safely be written to find the *first* of *all* the solutions. ~~This is related to the distinction between call by name and call by value in ALGOL68 and other languages [Stoy 68-9]~~.

The question now arises as to whether an expression may have two distinct normal forms, or two expressions with distinct meanings may be interconvertible[4]. Fortunately, the answer is *no* because of the <u>Church-Rosser theorem</u>: both (β) and ($\beta\eta$) satisfy the diamond property [Bar. 53, 63 & 65] that if M reduces to each of M_1, M_2 then they each reduce to some common M_3.

Some expressions, such as $(\lambda x.xx)(\lambda x.xx)$, have no normal form: in this case (β) gets us nowhere and (η) is inapplicable. Expressions like this are rather like *syntactic*

sinks: things can be substituted in them from outside, but they can never be broken up. We say that an expression has a <u>head normal form</u> if it is convertible to one like $\lambda \underline{x}.\, y M_1 M_2 \dots M_r$, and we may consider other expressions as *meaningless* [Bar. 41-3].

However the combinator

$$Y = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

is far from useless, since it yields (least) fixed points, as may be verified by reducing $F(YF)$. The *reason* (rather than the *proof*) why they are *least* is that (with $W = (\lambda x.\, f(xx))(\lambda x.\, f(xx))$) $Y = \lambda f.\, W \equiv \lambda f.\, fW \equiv \lambda f.\, f(fW) \equiv \dots \equiv \lambda f.\, f^{\infty}\bot$. In the $P\omega$ model, Y is in fact the least fixed point operator, and where this is so in general is essentially related to the notion of a <u>sensible</u> model, ie where all *meaningless* expressions (such as $(\lambda x.\, xx)(\lambda x.\, xx)$) are identified with $\bot$, being the least fixed point of the identity.

By repeatedly reducing expressions to some head normal form (or $\bot$ if none exists) we may represent a λ-expression by a <u>Böhm tree</u> whose leaves are variables or $\bot$. Böhm trees may be partially ordered by *amputation* and hence form an algebraic cpo, $\mathfrak{B}$, the finite trees being the compact elements. $\mathfrak{B}$ is then a λ-model.

In the lambda calculus we choose some sequence of combinators, $\ulcorner 0 \urcorner, \ulcorner 1 \urcorner, \ulcorner 2 \urcorner, \dots$, called <u>numerals</u> to represent the natural numbers, $\mathbb{N}$. These we require to be generated by $\ulcorner 0 \urcorner$ and **succ**, where **succ**$\ulcorner n \urcorner = \ulcorner n+1 \urcorner$. Then, to show that the λ-calculus is actually what is needed for computation, we have [Bar. 39-40] a function $f: \mathbb{N}^k \to \mathbb{N}$ is λ-definable, ie there's some combinator F such that $F \ulcorner n_0 \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_{k-1} \urcorner = \ulcorner f(n_0, \dots, n_{k-1}) \urcorner$, iff f is recursive, hence iff f is [Turing-]computable.

Two combinators of particular importance are

$$K = \lambda x.\lambda y.\, x \quad \text{and} \quad S = \lambda x.\lambda y.\lambda z.\, xz(yz)$$

since they generate all of the other combinators by application [Bar. 161-2]. If x, y are given types p, q then the type of K becomes

$$p \to (q \to p);$$

likewise if x, y, z have types $p \to (q \to r)$, $p \to q$ and p then

S has type

$$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$$

which, after de-currying and some rearrangement, becomes

$$p \wedge (p \to q) \wedge (p \wedge q \to r) \to r.$$

Both of these are well-known rules of propositional logic. Notice also that $I = \lambda x.x = SKK$. The combinators can in fact be generated by one element, $X = \langle K, S, K \rangle$, with $K = (XX)X$ and $S = X(XX)$.

As a practical demonstration of the validity of the foregoing remarks, Clarke et al. [1980] have built a machine called SKIM which uses the S, K, I combinators in place of a more conventional machine code instruction set.

$\mathcal{B}$, the set of Böhm trees, affords one model of the λ-calculus in terms of an algebraic cpo, but we shall consider the *graph* model $P\omega$, which (as previously remarked) is an algebraic *lattice*. The advantage[5] is not theoretical, merely that we have a theory of data types, which will be discussed in chapter III. The remainder of this chapter is due to Scott [1975, pp.522-534] but see also [Bar. 467-475] and [Stoy 115-132] for alternative accounts.

Here $\omega = \{0,1,2,\ldots\}$ and $P\omega$ is its powerset; write $P_f\omega$ for the (countable) set of finite subsets of ω. Scott interprets elements of $P\omega$ as *multiple integers*[6], writing, eg, $\{0,5,17\} = 0 \cup 5 \cup 17$, where the constituent parts *cooperate*. Thus $\emptyset = \bot$ and $\omega = \top$ in contrast to the domain of reals, $D_{\mathbb{R}}$; the distinction in meaning between these two examples will be discussed further in chapter VI.

$P\omega$ is given the Scott topology, which is the ω-fold product on $2 = \{0,1\}$ not with the *discrete* topology, $\{\emptyset, \{0\}, \{1\}, \{0,1\}\}$ but the Sierpinski topology, $\{\emptyset, \{1\}, \{0,1\}\}$. Thus in $P\omega$ the open sets are those of *finite character*, so $x \in U$ iff some finite $y \subseteq x$ has $y \in U$ and a function is continuous iff it preserves directed sups, so it's given by its values on finite elements $x \in \omega$ (on which it need only be monotone).

We need to introduce *explicit* bijections

$$\omega \times \omega \cong \omega \quad \text{by} \quad (n,m) \mapsto \tfrac{1}{2}(n+m)(n+m+1)+m$$

$$\omega \cong P_f\omega \quad \text{by} \quad n \mapsto \{k_i\} = e_n \text{ where } k_0 < k_1 < \ldots < k_{m-1} \text{ and } n = \sum_{i<m} 2^{k_i}$$

so that pairs and finite subsets may be canonically identified with integers. Then $f : P\omega \to P\omega$ is continuous iff

$$f(x) = \bigcup\{f(e_n) : e_n \subseteq x\}$$

[Scott 525] and is hence determined by

$$\mathrm{graph}(f) = \{(n,m) : m \in f(e_n)\}$$

since $\mathrm{func}(\mathrm{graph}(f)) = f$ where

$$\mathrm{func}(u) : x \longmapsto \{m : \exists e_n \subseteq x : (n,m) \in u\}$$

for $u \in P\omega$ Moreover $u \subseteq \mathrm{graph}(\mathrm{func}(u))$[7]

Thus graph and func are our λ, Ξ from the beginning of this chapter since they may be shown to be continuous functions $[D \to D] \to D$ and $D \to [D \to D]$ themselves, with $D = P\omega$. $P\omega$ is now a $\lambda\beta$-model since it satisfies $[(\alpha)_1]$ (β) and (ξ). For we have

$$u(x) = \{m : \exists e_n \subseteq x : (n,m) \in u\}$$

$$\lambda x.\tau(x) = \{(n,m) : m \in \tau(e_n)\}.$$

We now have that all LAMBDA-definable functions are continuous [Scott 529-530], and any continuous function $f : D^k \to D$ can be written as

$$f(x_0, \ldots, x_{k-1}) = u(x_0)(x_1)\ldots(x_{k-1})$$

for some $u \in P\omega$. Scott [531-9] then goes on to discuss the relationship to computability.

A set of integers is <u>recursively enumerable</u> if it is the range of some recursive (ie λ-definable) function. There are countably many such sets and they are closed under application and abstraction, hence forming a sub-λ-model of $P\omega$. They do not, however, form a cpo. This is exactly the sense in which we said π is computable in chapter I.

Concerning fixed points, the functional

$$\mathrm{fix} : [D \to D] \to D \quad \text{by} \quad f \mapsto \bigcup\{f^n(\bot) : n \in \omega\}$$

yields the least fixed point of f (by Tarski's theorem) and is itself continuous. The combinator Y also yields fixed points and *morally*[8] ought to give the *least* such. In our model it does in fact [ie $Y(\mathrm{graph}(f)) = \mathrm{fix}(f)$] but the proof uses the explicit bijections $\omega \times \omega \cong \omega \cong P_f\omega$ chosen above [Scott 531 and 569-70]; the same also holds in Scott's D_∞ models [Bar. 493-4] but this again depends on the particular projections chosen. D_∞ will be discussed in chapter IV.

Why should we want *least* fixed points anyway?[9] Consider Stoy's [79-80] example:

$$f(x) = \begin{cases} 1 & x = 0 \\ f(3) & x = 1 \\ f(x-2) & \text{otherwise} \end{cases}$$

for which one would naturally say,

$$f(x) = \begin{cases} 1 & x \text{ even} \\ \text{undefined} & \text{otherwise} \end{cases}$$

but one may equally well give any arbitrary value a for odd arguments, so why should one solution be better than any other? Clearly we choose the first because it *says no more* than the original definition, ie it is the *least* solution.

The λ-calculus is very weak on the prescription of equality. For instance, the functions

$f(n) = n$ and $g(n) =$ **if** $n=0$ **then** 0 **else** $g(n-1)+1$

considered as $f, g : \mathbb{N} \to \mathbb{N}$ are clearly equal, but they are not interconvertible. In this sense the λ-calculus fails to describe *computations* (at least of expressions without a normal form) since it is possible for two expressions to behave identically but yet for one of them not to have a normal form. Wadsworth [1976] discusses a notion of <u>approximate normal form</u> and how D_∞ provides exactly what is required.

The justification for using $P\omega$ here lies in its *topological* and *categorical* properties, which will be discussed in the next chapter. First, every countably-based[10] T_0-space can be embedded in $P\omega$ by

$$x \mapsto \{n : x \in U_n\}$$

where $\{U_n : n \in \omega\}$ is the basis. Second, $P\omega$ is <u>injective</u>[11], ie if $f : X \to P\omega$ is a continuous map and $X \subseteq Y$ (as a subspace), then there's some $g : Y \to P\omega$ with $g|_X = f$ [Scott 527]. Thus assuming our domain to be a countably-based T_0 topological space, we may restrict our attention to subspaces of $P\omega$; moreover any continuous map $D \to D'$ with $D, D' \subseteq P\omega$ may be assumed defined on $P\omega \to P\omega$. We may view the choice of a *topological* space as being forced on us by the necessity to cut down the function space (for cardinality reasons) in order to get a λβ-model.

$P\omega$ is not alone in being injective [Compendium II§3]. For the Sierpinski space, $2 = \{0,1\}$ with topology $\{\emptyset, \{1\}, \{0,1\}\}$, is easily shown to be so [*ibid.* II 3.3]. Moreover the Scott and product topologies on 2^κ (where κ is an arbitrary infinite cardinal) agree and are injective. Finally, every injective space, X, is a retract of such a powerset, ie there's some continuous $f : 2^\kappa \to 2^\kappa$ with $f^2 = f$ and $\mathrm{im} f \cong X$. Such retracts (as lattices) also classify the continuous lattices.

CHAPTER III THE RETRACT CALCULUS OF DATA TYPES

In this chapter a theory of data types is introduced. Typed λ-calculus was mentioned at the beginning of chapter II; its typing rules are so strong that there is no facility for the manipulation of types or conversion between them: any attempt at type *testing* results in type *breaking*. The *internal* type theory of this chapter, on the other hand, even has a (limited) *type of types*. The main sources are Scott [1976] and the Compendium [ie Gierz *et al.* 1980].

At the end of the previous chapter we found that a topological space is [T_0 and] injective iff it is a retract of some powerset with the Scott topology. It is also true [Comp. I 4.16] that a lattice is continuous iff it is a retract of some powerset under inclusion. This result enables us to find a *copy* of the category of countably based continuous lattices *inside* $D = P\omega$ by considering its (monoid of) endomorphisms.

It is of course not just a coincidence that continuous lattices and injective T_0 spaces should have the same classification. The topology arising from the lattice is the Scott topology and the order on the topological space is the <u>specialisation order</u>, $x \leq y$ if $(x \in U \Rightarrow y \in U)$ for all open sets U. This is antisymmetric (ie $x \leq y \leq x \Rightarrow x = y$) iff the space is T_0 and discrete iff it's T_1. The Scott topology and the specialisation order give rise to an equivalence between continuous lattices and injective T_0 spaces [Comp. II 3.8]. For further information see also Scott [1972] and Johnstone [1983 II §1].

For any monoid (here the function space $[P\omega \to P\omega]$ under composition), we have the <u>Karoubian</u> category $K(M)$ whose objects are the idempotents $a \in M : a^2 = a$ and whose morphisms $u : a \to b$ are elements of M such that $au = u = ub$. The properties of $K(M)$ will be discussed further in the next chapter. We shall write $\mathcal{C}_\omega = K([P\omega \to P\omega])$ for the category which arises thus from $P\omega$ and <u>CONT_ω</u> for the category of countably-based continuous lattices and Scott-continuous maps.

These two categories, $\mathcal{C}_\omega$ and $\underline{\text{CONT}}_\omega$, are equivalent, and also cartesian closed. First, for $a \in \mathcal{C}_\omega$ (ie $a: D \to D$ st. $a^2 = a$), put $\tilde{a} = \text{im}\, a = \{x \in D : x^a = x\}$, which is a continuous lattice [Comp. I 2.14] which is clearly countably-based. Conversely, every countably-based continuous lattice arises (up to isomorphism) in this way [Comp. I 4.16], so the functor $\mathcal{C}_\omega \to \underline{\text{CONT}}_\omega$ by $a \mapsto \tilde{a}$ and $u \mapsto \tilde{u} = u|_{\tilde{a}}^{\tilde{b}}$ (where $|^{\tilde{b}}$ denotes corestriction) is essentially surjective. It is also full and faithful, for if $f: \tilde{a} \to \tilde{b}$ in $\underline{\text{CONT}}_\omega$, put $u = p_A f i_B$ (where $p_A = a : D \twoheadrightarrow \tilde{a}$ and $i_B = b : \tilde{b} \hookrightarrow D$ are the natural projection and embedding maps) then $\tilde{u} = f$ as required.

Now to cartesian closure, first of $\underline{\text{CONT}}_\omega$ (of course since this is a *categorical* property and $\mathcal{C}_\omega \simeq \underline{\text{CONT}}_\omega$ we only need to check one, but we shall discuss both). Given two continuous lattices, A, B, the Scott-continuous maps between them are partially ordered pointwise, and as such form a continuous lattice, $[A \to B]$. The proof of this will be given in chapter IV. The initial object is the trivial (one-point) lattice and the product is the cartesian product with the componentwise order; conveniently (in contrast to the well-known examples in $\mathbb{R}^2$) componentwise and joint continuity coincide [Comp. II 2.9]. We then have the isomorphism

$$\text{curry} : [A \times B \to C] \longrightarrow [A \to [B \to C]]$$

which renders $\underline{\text{CONT}}_\omega$ cartesian closed [Comp. II 2.10]. Of course we also have

$$\text{eval} : [A \to B] \times A \longrightarrow B$$

as a continuous map.

In order to do the same thing within $P\omega$ we first need the technical devices of conditionals and pairs.

$$\textit{if } z \textit{ then } x \textit{ else } y = \begin{cases} \bot & \text{if } z = \bot \\ x & \text{if } z = 0 \\ y & \text{if } 0 \notin z \neq \bot \\ x \cup y & \text{if } 0 \in z \neq 0 \end{cases}$$

$$\langle x, y \rangle = \lambda z.\ \textit{if } z \textit{ then } x \textit{ else if } z{-}1 \textit{ then } y \textit{ else } \bot$$

The latter works for our present purposes, but later when we discuss closure operators we shall need instead

$$\langle x, y \rangle = \{2n : n \in x\} \cup \{2m{+}1 : m \in y\}$$

$$u_0 = \{n : 2n \in u\} \qquad u_1 = \{m : 2m{+}1 \in u\}$$

Now we define the product of two retracts as

$$a \times b = \lambda u.\ \langle a(u_0), b(u_1) \rangle$$

and the function space as

$$[a \to b] = \lambda u. \lambda x.\ b(u(a(x))).$$

These functions have all the obvious properties which we would expect of them [Scott 542-4] and render $\mathcal{C}_\omega$ cartesian closed. More importantly (as is fairly clear) they correspond directly to the product and function space already defined in CONT$_\omega$, for

$$(a\times b)^{\sim} \cong \tilde{a}\times\tilde{b} \qquad [a\to b]^{\sim} \cong [\tilde{a}\to\tilde{b}]$$

We may also define sums of retracts (or continuous lattices). The *categorical* sum (coproduct) is the disjoint union of the two lattices, but with top and bottom respectively identified. This is called the coalesced sum and is *not* what we usually require. For if $x=\bot_A \in A \subset A+B$, we do at least know of x that it's in A, so it should be distinct from $\bot_B$ (which we know to be in B) and $\bot$ (about which we know nothing. Accordingly we define $A+B$ to have an extra top and bottom added to it. In $P\omega$ we put

$$a+b = \lambda u.\ \textit{if } u_0 \textit{ then } \langle 0, a(u_1)\rangle \textit{ else } \langle 1, b(u_1)\rangle$$

Unfortunately $(A+B)+C$, $A+B+C$ and $A+(B+C)$ are all distinct, the choice being made *ad hoc*.

We have an extra partial ordering on $\mathcal{C}_\omega$ essentially given by inclusion in CONT$_\omega$. Write $a \preccurlyeq b$ for $a = a\circ b = b\circ a$ [Scott 541]. A retract with $\bot \preccurlyeq a$, ie $a:\bot\mapsto\bot$, is called strict; clearly $\bot$ is the $\preccurlyeq$-least strict retract and 1 is the $\preccurlyeq$-greatest since $\tilde{1}\cong D$.

The interpretation of this in programming languages ties up the notation rather neatly if we write

$$x : a \iff a(x)=x \iff x = a(y)\ (\exists y)$$

for the statement that x has type a, as in, say, PASCAL. Alternatively we may think of it as $x := a(x)$, ie *whatever x is, reduce it to type* a. Then we have typed λ-expressions like

$$\lambda x{:}a\,.\,f(x){:}b \quad = \quad \lambda x.\ b(f(a(x)))$$

for the function $f:\tilde{a}\to\tilde{b}$. Also

$$f : [a\to b] \iff f:\tilde{a}\to\tilde{b}$$

where the notation on the right is the usual category-theoretical one for maps between objects (continuous lattices).

Because of Scott's argument (chapter I) and the equivalence of $\mathcal{C}_\omega$ and CONT$_\omega$, we identify data types and retracts. Examples include[2]

$$\textbf{int}:\ x \longmapsto \begin{cases} x & \text{if } \operatorname{card}(x)\le 1 \\ \top & \text{otherwise}\end{cases}$$

$$\mathbf{bool}: x \longmapsto \begin{cases} \bot & \text{if } x=\bot \\ 0 & \text{if } x=0 \\ \{1,2,\dots\} & \text{if } 0\notin x\neq\bot \\ \top & \text{otherwise} \end{cases}$$

and some representation for the type **real** which could be worked out with patience and ingenuity.

The value of this notation and definition is that we may solve *domain equations* such as

$$\text{tree} = \mathbf{int} + \text{tree} \times \text{tree}$$

which defines the type of binary trees with integer leaves. The solution of this is the least fixed point of the continuous function

$$f : t \longmapsto \mathbf{int} + t \times t$$

the existence of which is guaranteed by the fixed point theorem (using Y) and the fact that if f preserves retracts, $\mathrm{fix}(f) = \bigvee\{f^n(\bot) : n\in\omega\}$ is also a retract. [Scott 545-6], for which we use the Scott-continuity of $\mathrm{eval} : [D\to D]\times D \to D$. In fact we have much more than this: if f is $\trianglelefteq$-monotone and preserves strictness, $Y(f)$ actually performs the inverse limit construction of chapter IV. These properties do in fact hold of $+$, $\times$ and $\to$. When we apply the theory to programming languages in chapter V we shall make frequent use of such domain equations.

Now consider the question of the existence of a type of types. This actually turns out not to be possible, but not (apparently) because of any Russell-style paradox. The function

$$f : u \longmapsto u \circ u$$

is continuous (and strict) and its fixed points (which form a complete lattice) are clearly the retracts. However it is not *itself* a retract since it's not idempotent. In fact there is *no* retract whose image is the space of retracts since they do not form a continuous lattice. The proof is due to Y. L. Ershov [Scott 574; Bar. 492]

All is not, however, lost, for we can reformulate the theory for algebraic rather than continuous lattices. Unfortunately we thereby lose the domain of reals since $D_{\mathbb{R}}$ is continuous but not algebraic [*supra* p.3]. Perhaps there might be some suitable intermediate class, but it seems unlikely. Scott [559-565] also gives a construction of yet another cartesian closed category of partial equivalence relations on $P\omega$ in which $\mathscr{C}_\omega$

is embedded as a full subcategory.

A <u>closure operator</u> is a retract with $a \geq 1$. The cartesian closed category $\underline{ALG}_\omega$ of countably-based algebraic lattices is equivalent to the full subcategory of $\mathcal{C}_\omega$ of closure operators, essentially by [Comp. I 4.15 and II 2.8]. Algebraic lattices arise naturally as subobject lattices in groups, rings, etc. [Comp. pp. XI to XIX and p. 95]. It is largely algebraic cpos that are used for semantics: $\mathcal{B}$, the space of Böhm trees, is one for example [*supra* p. 9].

The definitions of $+ \times$ and $\rightarrow$ require slight modification, as has already been noted [*supra* p. 13 and Scott pp. 549-551]. The fixed-point combinator, Y, has the desired properties under the additional assumption that $f(\bot) \geq 1$ since $\bot$ is not itself a closure operator. Alternatively we may use $\bigvee\{f^n(1) : n \in \omega\}$ instead.

The closure operators themselves now form an algebraic lattice, given by the closure operator

$$V = \lambda a.\lambda x.Y(\lambda y.\ x \cup a(y)) : a \mapsto \bigvee\{a^n : n \in \omega\}$$

[Scott 551-2]. Thus V is the required type of types. It may be viewed as the *internal* category $\underline{ALG}_\omega$ and the functors $+, \times : \underline{ALG}_\omega \times \underline{ALG}_\omega \rightarrow \underline{ALG}_\omega$ and $[\cdot \rightarrow \cdot] : \underline{ALG}_\omega^{op} \times \underline{ALG}_\omega \rightarrow \underline{ALG}_\omega$ become continuous maps $V \times V \rightarrow V$ and hence themselves *elements* of $P\omega$. Similarly $Y : [D \rightarrow D] \rightarrow D$.

Certain subcategories of $\underline{ALG}_\omega$ arise as ε-subretracts of V and functors as maps $V \rightarrow V$. The category of such categories is given inside $P\omega$ by the closure operator

$$V_1 = \lambda a.Y(\lambda x.V(a \cup V \circ x \circ V)) = \bigvee\{V^n \circ V(a) \circ V^n : n \in \omega\}$$

Of course we can play this trick repeatedly. I do not, however, know the characterisation of the subcategories of $\underline{ALG}_\omega$ which arise.

The usefulness of V and V_1 is that we have a systematic way of dealing with families of operations which are defined with *parametric* types. Take, for example,

$$eval_{A,B} : [A \rightarrow B] \times A \rightarrow B.$$

In order to use this we have to define a separate operator for *each* pair of types (A, B). Instead we may define

$$eval : V \times V \rightarrow [[D \rightarrow D] \times D \rightarrow D]$$

by

$$eval(A,B) : \langle f,x \rangle \longmapsto ([A \to B](f))(A(x))$$

and of course there are more sophisticated examples using categories and V_1 in place of domains and V. Such higher types we may describe as <u>type schemes</u>.

Thus in $P\omega$ we have provided not only a model of the untyped $\lambda\beta$-calculus suitable for computation, but also a systematic theory of data types – both atomic and structured – and a means of dealing with them inside the semantics. In the next chapter we shall pursue in detail the category theory which we have employed.

CHAPTER IV SELF-ACTING MONOIDS

In this chapter some of the constructions mentioned previously will be reviewed functorially and some properties of monoids will be discussed. A suitable category of lattice models for the λ-calculus will be sought and Scott's D_∞ model will be constructed. Finally the method will be generalised in a manner applicable to multiple-process semantics which is the subject of chapter VI.

A monoid, M, may be regarded as a category with one object. Such a category is unlikely to have equalisers and coequalisers [MacLane 1971 pp20,70,etc.], so it is unlikely that an idempotent, $e \in M$ st. $e^2=e$, will split, ie' $e=pi$ where $ip=1$. However, by the construction mentioned before [*supra*. p.12], we have a category $K(M)$, whose objects are the idempotents of M, in which this does occur. Moreover the construction is universal (and functorial) in the sense that if A is another category with split idempotents and $F: M \to A$ is a functor then there is a unique $\bar{F}: K(M) \to A$ with $\bar{F}|_M = F$, where $M \hookrightarrow K(M)$ by $\bullet \mapsto 1_M$ (which is a full embedding).

Since K is the restriction to monoids of the left adjoint to the inclusion of categories with split idempotents into all categories, K preserves colimits [MacLane 114-5]. Morphisms of monoids *quā* categories are just monoid homomorphisms (ie preserving identity and composition); these are mapped faithfully to functors *preserving* $\mathcal{U}=1_M$. General functors are exactly the images of *semigroup* homomorphisms (ie just preserving composition). Thus in this context we should not require the identity to be preserved: merely to exist. Since our monoids arise as endomorphism spaces of objects this is not surprising, since we should not expect the identity map on an object to be sent by a non-surjective map to the identity on an image.

$K(M)$ need not be a *skeletal* category: there may be distinct isomorphic objects $e,f \in K(M)$. This arises iff there are $u,v \in M$ with $uv=e$, $vu=f$, $uvu=u$ and $vuv=v$. Since *equivalence* arises more naturally in categories than *isomorphism*, we must employ these notions translated

back from categories to monoids. It's not clear what (if any) universal property $U = 1_M$ satisfies in $K(M)$ which would enable us to convert equivalence into isomorphism for some class of monoids, and thereby answer the question of whether $P\omega$ is unique as a *universal* domain. [Smyth & Plotkin 1978, pp. 28-29].

Given an endomorphism monoid, in what circumstances can one recover the original object? If it's a topological space, always. For the points of the space betray themselves as (the images of) *constant* maps, ie those such that $mc = c$ for all $m \in M$. For some purposes, such as the same question applied to vector spaces, it is more convenient to replace $=$ by some other relation preserved by multiplication. We say that M has enough constants if

$$cm = cn \text{ for all constants } c \Rightarrow m = n.$$

The endomorphism monoid of a topological space (say with the pointwise topology) always has enough constants, and the subspace of constants is canonically homeomorphic with the given space. On the other hand, a compact Hausdorff topological monoid with enough constants can be embedded in the endomorphism space of its space of constants [easy proofs].

Now consider the lattice models of $\lambda\beta$ and $\lambda\eta$ [*supra* pp. 6-7] : what category do we have of them? For convenience, let us define a continuous $\lambda\alpha$-lattice[2] to be a triple $(D, \lambda, \bar{\varepsilon})$ consisting of a continuous lattice D and maps $\lambda : [D \to D] \to D$ and $\bar{\varepsilon} : D \to [D \to D]$, with $\varepsilon : D \times D \to D$ defined in the obvious way. Clearly we shall want our maps to be Scott-continuous and preserve $\bar{\varepsilon}$, λ and ε. However in order to get a map $[D \to D] \to [E \to E]$ we shall need maps both $\psi : D \to E$ and $\varphi : E \to D$ so that $(\psi, \varphi) : f \mapsto \psi f \varphi$; in fact we shall choose both $(\lambda, \bar{\varepsilon})$ and (ψ, φ) to be adjoint pairs.

Continuous $\lambda\beta$- and $\lambda\eta$-lattices are now defined as those with $\lambda\bar{\varepsilon} = 1$ and $\lambda\bar{\varepsilon} = 1$, $\bar{\varepsilon}\lambda = 1$ respectively. A projection pair $(\psi, \varphi) : D \to E$ has $\psi\varphi \leq 1$ and $\varphi\psi = 1$, so that ψ is left adjoint to φ, φ is injective and ψ is surjective. That they should be adjoint is justified by the functoriality of the function-space construction [Compendium IV §3] and the fact that the colimit of a system of φ's coincides with the limit of a system of ψ's.

$\varphi: E \to D$ is now obliged to preserve all infs as well as directed sups since it is a right adjoint [Compendium O§3], so that it is a homomorphism of continuous lattices considered as directed-distributive complete semilattices [Comp. I§2]. D,E can be given the Lawson topology [Comp. III§1], with respect to which $\varphi: E \to D$ is continuous iff it preserves infs and directed sups[3]. The condition on φ is now exactly that it be a Lawson embedding which is also a homomorphism of $\lambda, \bar{\varepsilon}, \varepsilon$, etc. [There is an error of definition here

The Lawson topology is the coarsest topology in which Scott-open sets remain open and all principal filters (ie sets of the form $\uparrow x = \{y \geq x\}$ for $x \in D$) are closed. This is then compact Hausdorff [Comp. III 1.10] and conversely a complete lattice in which binary meet is Scott-continuous and which is Hausdorff in its Lawson topology is continuous [*ibid.* 2.9]. Moreover the equivalent extra conditions of total disconnectedness and zero-dimensionality are exactly what is required to render the lattice algebraic and the topology a Stone Space [ibid. 2.16 and Johnstone 1983 §2.4].

Why do we need φ to be an embedding (or ψ a projection)? Because a general Lawson-continuous $(\lambda, \bar{\varepsilon})$-homomorphism does not preserve composition, for which we require, for $f, g: E \to E$,

$$\psi f \varphi \psi g \varphi = \psi f g \varphi$$

Considering constant f, this is equivalent to $\varphi\psi g\varphi = g\varphi$, for which it is clearly *sufficient* that $\varphi\psi = 1$, or alternatively that φ be constant and hence map the whole of E to the trivial lattice $\{\top\} \subseteq D$. We may exclude the latter case, whence it will be shown that $\varphi\psi = 1$ is also *necessary*.

This curious condition has a partial explanation in the λ-calculus. For if we think of our model as being *generated* by those terms with a normal form, it reduces to the condition that we cannot identify any two distinct normal forms without identifying everything, a fact which was shown by Böhm [Bar. 252]. It seems remarkable that the same condition should arise from considering monoids, λ-calculus and the Scott construction. However it would be wrong to require (ψ, φ) also to preserve the identity, for the reasons given at the beginning of this chapter and also that then $\psi\varphi = 1$, so (ψ, φ) is an isomorphism.

It is convenient here to give the long-awaited proof that $[D \to D']$ is *actually* a continuous lattice, since the method also shows the necessity of φ being injective [Comp. II 2.16 and IV 4.23]. For $x \in D, y \in D'$ define $[x \Rightarrow y] : D \to D'$ by $z \mapsto y$ if $x \ll z$ and $\bot$ otherwise; this is clearly Scott continuous since it's essentially the (Sierpinski) characteristic function of the Scott-open set $\Uparrow x$. Now if $f : D \to D'$ and $y \ll f(x)$ in D' we have $[x \Rightarrow y] \ll f$ in $[D \to D']$ since for directed $S \subseteq [D \to D']$ $\sup S \geq f$ iff $\sup\{s(z) : s \in S\} \geq f(z)$ for all z: whence $s(x) \geq y$ for some $s \in S$ and $s(z) \geq s(x) \geq y$ for $x \ll z$ so $s \geq [x \Rightarrow y]$ as required. Moreover $f(z) = f(\sup_x \{x : x \ll z\})$ $= \sup_x \{f(x) : x \ll z\} = \sup_{x,y} \{y \leq f(x) : x \ll z\} = \sup_{x,y} \{[x \Rightarrow y](z) : y \leq f(z)\}$, so f is the sup of the set $\{[x \Rightarrow y] : y \leq f(z)\}$ and hence of the larger set $\{g : g \ll f\}$. $[D \to D']$ is thus a continuous lattice. Now to the necessity of $\varphi\psi = 1$ in order that $\varphi\psi g \varphi = g\varphi$; since g is a sup of functions of the form $[x \Rightarrow y]$ and everything is Scott-continuous, it suffices to check it for such functions and non-constant φ. But $z\varphi\psi[x \Rightarrow y]\varphi = y$ iff $x \ll z\varphi\psi$ and $z[x \Rightarrow y]\varphi = y$ iff $x \ll z$ so we require $x \ll z\varphi\psi \Leftrightarrow x \ll z$. But then since D' is a continuous lattice this holds iff $z = z\varphi\psi$ for all z, as required. We've also shown that $[D \to D']$ is countably based if D, D' are.

We shall now define the categories $\underline{\lambda\alpha\text{-CL}}$, $\underline{\lambda\beta\text{-CL}}$ and $\underline{\lambda\eta\text{-CL}}$ of continuous $\lambda\alpha$, $\lambda\beta$ and $\lambda\eta$ lattices as being the triples $(D, \lambda, \bar{\varepsilon})$ as above with D nontrivial, with morphisms the *projection*-pairs (ψ, φ), so that the arrow $\varphi : E \to D$ with which we have been working is in fact *contravariant*. Observe that $\underline{\lambda\eta\text{-CL}} \subset \underline{\lambda\beta\text{-CL}} \subset \underline{\lambda\alpha\text{-CL}}$ as *full* subcategories; in fact we shall show that $\underline{\lambda\eta\text{-CL}}$ is a coreflective[4] subcategory of $\underline{\lambda\alpha\text{-CL}}$ and hence also of $\underline{\lambda\beta\text{-CL}}$. We may similarly define the categories $\underline{\lambda\alpha\text{-AL}}$, $\underline{\lambda\beta\text{-AL}}$ and $\underline{\lambda\eta\text{-AL}}$ of *algebraic* $\lambda\alpha$, $\lambda\beta$ and $\lambda\eta$ lattices, for which the same is true.

Each of these categories has filtered colimits and cofiltered limits, essentially because the function-space functor preserves them (and also injectivity and surjectivity) [Compendium IV 3.19]. The cofiltered limit of the ψ's is also the filtered colimit of the φ's: this *limit-colimit coincidence* was first remarked by Scott [1972]. If an object is the cofiltered limit of *finite* $\lambda\alpha$-lattices (which are then trivially algebraic), we shall call it <u>profinite</u>[5]. There are, of course, no finite $\lambda\beta$- or $\lambda\eta$-lattices (by cardinality, since there are nonconstant functions as well as constant ones with each value).

The function-space functor, $\text{Funct} : \underline{CL} \to \underline{CL}$ [Comp. IV §3] may be extended to $\underline{\lambda\alpha\text{-}CL} \to \underline{\lambda\alpha\text{-}CL}$ in an obvious way using pre- and post-composition with $\lambda, \bar{\varepsilon}$. It may be verified that it restricts to $\underline{\lambda\beta\text{-}CL}$, $\underline{\lambda\eta\text{-}CL}$, $\underline{\lambda\alpha\text{-}AL}$, $\underline{\lambda\beta\text{-}AL}$ and $\underline{\lambda\eta\text{-}AL}$, although of course on $\underline{\lambda\eta\text{-}CL}$ and $\underline{\lambda\eta\text{-}AL}$ it's the identity. It is not, however, the required adjoint to the inclusion of $\underline{\lambda\eta\text{-}CL}$ since it's not idempotent.

If, however, we take a $\lambda\alpha$-lattice D with $\bar{\varepsilon}\lambda = 1$, we already have a projection pair $(\lambda, \bar{\varepsilon}) : \text{Funct}(D) \to D$. Repeated application of Funct yields a projective system

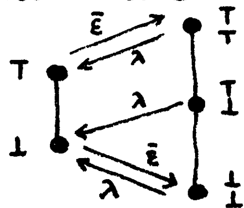
$$D \xleftarrow{(\lambda,\bar{\varepsilon})} \text{Funct}(D) \xleftarrow{\text{Funct}(\lambda,\bar{\varepsilon})} \text{Funct}^2(D) \longleftarrow \cdots \quad \widetilde{\text{Funct}}(D)$$

whose limit is a $\lambda\eta$-lattice since Funct preserves such limits [Comp. IV §4]. $\widetilde{\text{Funct}}$ is then the right adjoint to the inclusion (a cone over this diagram may be built up from $(\psi, \varphi) : E \to D$ by post-composition with $\text{Funct}^n(\bar{\varepsilon})$). The condition that $\bar{\varepsilon}\lambda = 1$ may be dropped by taking $D_n = \text{im}\, \bar{\varepsilon}_{n-1}\lambda_{n-1} \subseteq [D_{n-1} \to D_{n-1}]$; for a $\lambda\beta$-lattice this amounts to taking repeatedly the retraction to the functionspace, which is essentially what we get by solving the (algebraic) domain equation

$$D_\infty = I + [D_\infty \to D_\infty]$$

in $P\omega$, to which we shall return shortly.

Scott's original D_∞ model was obtained from the two-element $\lambda\alpha$-lattice.



Here $\lambda : f \mapsto f(\bot)$ and $\bar{\varepsilon} : x \mapsto c_x$, the constant function with value x. Any continuous $\lambda\alpha$-lattice *with these* $\lambda, \bar{\varepsilon}$ may be used to give an essentially similar [Wadsworth 1976, pp. 513-4] model, in which $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is the *least* fixed-point operator. If some other choice is used this may no longer hold [*ibid.* and Bar. 493-4]. Funct does *not* however preserve this *definition* of $(\lambda, \bar{\varepsilon})$, as may be seen by close examination of Stoy's [114] diagram of D_0, D_1, D_2 under this construction. Thus it is not obvious how to characterise such *sensible* $\lambda\eta$-models.

Let us now return to $P\omega$ and show that the same results may be obtained using fixed-points in the retract calculus. [Scott 545]

Suppose f preserves $\preccurlyeq$ and strict retracts; then the range of $c = Y(f)$ is isomorphic to the limit of the ranges of the (strict) retracts $f^n(\bot)$ for $n \in \omega$ [Scott 545]. For by induction $f^n(\bot) \preccurlyeq Y(f)$ and the retracts $f^n(\bot)$ are the projections of c onto the terms $c_n = f^n(\bot)$, and the $u : c$ map isomorphically to the sequences $\langle u_0, u_1, \ldots \rangle$ with $u_n : f^n(\bot)$ and $u_n = u_{n+1} f^n(\bot)$ as required for universality.

For the algebraic case the same applies so long as either $f(\bot) \geq 1$ ($\bot$ is not a closure operator) or we redefine the fixed-point operator as $\bigvee\{f^n(1) : n \in \omega\}$. Thus in particular taking $f = \lambda a{:}V.\,[a \to a]$, $Y(f)$ gives the limit of the sequence

$$\bot,\ I,\ [I \to I],\ [[I \to I] \to [I \to I]],\ [[[I \to I] \to [I \to I]] \to [[I \to I] \to [I \to I]]], \ldots$$

which is clearly an algebraic $\lambda\eta$-lattice since $a \mapsto [a \to a]$ is a continuous closure operator. This gives the same result as the reflection of $P\omega$ into $\underline{\lambda\eta\text{-AL}}$ above.

We may also mimic the D_∞ construction starting from the two-element $\lambda\alpha$-lattice. Let $a_0 : P\omega \to P\omega$ by $\bot \to \bot$ and $x \neq \bot \to \top$; this is a closure operator with the required image. Now let $a_{n+1} = [a_n \to a_n]$, giving a *decreasing* sequence with $a_\infty = \inf a_n$. We must consider $a_n \in [D \to D] \subset D$ and take the inf as such; fortunately λ (but not $\bar{\varepsilon}$) preserves all infs since $\lambda \dashv \bar{\varepsilon}$ so $\inf_{[D \to D]} a_n = \inf_D a_n$, and $a_\infty \in [D \to D]$. a_∞ is bounded below by the limit of the previous problem, so $a_\infty \geq 1$, but it's not clear that $a_\infty a_\infty = a_\infty$ or that $a \mapsto [a \to a]$ is Lawson continuous.[6]

Smyth & Plotkin [1978, 1982] have generalised the D_∞[7] construction by working in categories enriched over $\underline{\text{cpo}}$. Thus the hom-sets are cpos (posets with bottom and directed sups) and all functors (in particular composition) are strict and Scott-continuous (they *preserve* bottom and directed sups). As before, we're particularly interested in projection pairs.[8]

The advantage of such a category, K, is that we can often determine cofiltered limits *locally*. For if we have a cofiltered diagram $d : \mathcal{I} \to K$ of projection pairs and a cone $(\psi, \varphi) : X \to d$ over it (so that φ is a cone for the embeddings and ψ for the projections) then

$$d_\varphi : \mathcal{I} \to \mathrm{Proj}(X, X) \quad \text{by} \quad i \mapsto \psi_i \varphi_i$$

is a directed set in $\mathrm{Proj}(X,X)$ whose sup is the identity iff X is the limit of d (the limit of the ψ's and the colimit of the φ's).

Thus if we have split idempotents in $\underline{K}$ (which is equivalent to colimits of all finite filtered diagrams or limits of all finite cofiltered ones), then from any cone over the diagram we can construct a *universal* one. In particular, if $\underline{K} = K(M)$ for some ordered monoid M this is always possible. Since Scott's $\propto$ relation corresponds directly to the existence of an embedding (*aliter* projection), this explains why these limit-colimit constructions may be effected so easily within $P\omega$. Notice that every functor automatically preserves limits of projection pairs.

We may assume without loss of generality that $\underline{K}$ has a zero object, O, (although we deliberately excluded it from $\underline{\lambda\alpha\text{-CL}}$). If we also have limits of ω^{op}-chains of projection pairs, every endofunctor $F: \underline{K} \to \underline{K}$ has an *initial* F-algebra (A, α) given, as before, by the limit:

$$
\begin{array}{ccccccccc}
 & \overset{F}{\nearrow} & OF & \xleftarrow{\perp F} & OF^2 & \xleftarrow{\perp F^2} & \cdots & \leftarrow & O\tilde{F}F \cong O\tilde{F} \\
 & & \| & & \| & & & & \alpha \downarrow \cong \\
O & \xleftarrow[\perp]{} & OF & \xleftarrow[\perp F]{} & OF^2 & \xleftarrow[\perp F^2]{} & \cdots & \leftarrow & O\tilde{F} = A
\end{array}
$$

This is then the natural generalisation of the fixed-point theorem to categories.

In the next chapter we shall break from category theory to see how the Scott-Strachey theory is actually applied to programming languages. Then in the final chapter the extension of the theory to parallel processing will be discussed, which will bring us back to the work of Smyth & Plotkin.

CHAPTER V THE SEMANTICS OF PROGRAMMING LANGUAGES

The purpose of this chapter is to sketch *very* briefly how the foregoing theory is applied to actual programming languages. The main source is Stoy [1977] but shorter, earlier and more readable accounts are given in Scott & Strachey [1971], Strachey [1971] and Strachey & Wadsworth [1974]. It would, however, be inappropriate in a *mathematical* account to go into too much detail of the (rather difficult) *computer science* involved[1].

It has already been explained how one deals with recursively-defined functions and **while** loops as solutions of fixed-point equations. It is somewhat ironic that to us these are the *simplest* concepts and jumps and assignments present great difficulties (as we shall see), whereas the latter formed the *bread and butter* of the 1950s FORTRAN programmer[2], to whom the former were forbidden. In some sense the Strachey theory demonstrates the *formal complexity* associated with the *intuitive confusion* arising from a *spagghetti-like* program riddled with **goto**s[3].

The first problem is how to deal with stored variables (corresponding to free variables in λ-calculus). The <u>store</u>[4] (or <u>memory</u>) is a function from locations or addresses (alternatively names or identifiers: we shall make the distinction later),

$$\rho : L \to V$$

These are of course assumed to be Scott domains. Insulating text of the object language by means of fancy brackets, we write $\rho[\![I]\!]$ for the value of the identifier I in the environment ρ. Also $\rho[I:=\delta]$ denotes the environment obtained from ρ by changing the value of I to δ.

Now we turn to the interpretation of a λ-expression E, possibly containing free variables. This will be a function taking an environment to yield a value:

$$\mathcal{E}[\![E]\!] : [L \to V] \to V$$

so that $\mathcal{E}[\![E]\!]\rho$ denotes the value of an expression with its

free variables bound in. $\mathcal{E}$ itself is a (continuous) function whose domain is the *syntactic category*[5] of well-formed expressions, a subset of the space of strings of characters defined recursively in Backus-Naur form thus:

<Exp> ::= <Ide> | (λ.<Ide>.<Exp>) | (<Exp><Exp>)

corresponding to the domain equation

$$\mathrm{Exp} = \mathrm{Ide} + \mathrm{Ide} \times \mathrm{Exp} + \mathrm{Exp} \times \mathrm{Exp}.$$

Of course since Exp is defined recursively, so must be $\mathcal{E}$, by equations

$$\mathcal{E}[\![I]\!]\rho = \rho[\![I]\!]$$
$$\mathcal{E}[\![(\lambda I.E)]\!]\rho = \lambda \varepsilon : V.\ \mathcal{E}[\![E]\!](\rho[I := \varepsilon])$$
$$\mathcal{E}[\![(E_0 E_1)]\!]\rho = (\mathcal{E}[\![E_0]\!]\rho : [V \to V])(\mathcal{E}[\![E_1]\!]\rho : V)$$

whose consistency with the conversion rules it is now necessary to check. Observe that we have automatically taken care of the scope rules of λ-calculus: a variable is in scope for the duration of a λ-abstraction and becomes invisible for that of a nested one.

Next we treat the machine state, $\sigma \in S$, which can be modified by commands. For the moment we shall ignore the environment and assume commands *cannot* be stored. Commands are (semantically) changes of state, $\gamma : S \to S$; they have syntax

<Cmd> ::= <Primitive> | **if** <Exp> **then** <Cmd> **else** <Cmd> |
(<Cmd>;<Cmd>)

and there is an interpretation function $\mathcal{C} : \mathrm{Cmd} \to [S \to S]$, or rather $\mathcal{C} : \mathrm{Cmd} \to [[L \to V] \to [S \to S]]$, such that (for instance),

$$\sigma_0 \xrightarrow{\mathcal{C}[\![\Gamma_0]\!]\rho} \sigma_1 \xrightarrow{\mathcal{C}[\![\Gamma_1]\!]\rho} \sigma_2, \qquad \sigma_0 \xrightarrow{\mathcal{C}[\![\Gamma_0;\Gamma_1]\!]\rho} \sigma_2$$

commutes.

Now, since

$$\ulcorner \textbf{while}\ E\ \textbf{do}\ \Gamma \urcorner = \ulcorner \textbf{if}\ E\ \textbf{then}\ (\Gamma; \textbf{while}\ E\ \textbf{do}\ \Gamma)\ \textbf{else} \urcorner$$

we have

$$\mathcal{C}[\![\textbf{while}\ E\ \textbf{do}\ \Gamma]\!] = Y(\lambda\gamma.\lambda\sigma.\ \mathit{if}\ \mathcal{E}[\![E]\!]\sigma\ \mathit{then}\ \mathcal{C}[\![\Gamma]\!](\gamma\sigma)\ \mathit{else}\ \sigma)$$

Such constructs as $\ulcorner$**until** E **do** Γ$\urcorner$, $\ulcorner$**repeat** Γ **while** E$\urcorner$, etc. can be dealt with by means of the obvious transformations, which Stoy [207] calls *syntactic sugar*.

So far we have assumed that the evaluation of an expression causes no *side effect*. Of course a subroutine which evaluates an expression is entitled to make use of the machine's physical resources, but any resulting change must be *semantically*

undetectable, ie the language must not be sufficiently strong to probe, say, the machine stack. For instance, in the absence of any side effect we may say (as was erroneously required in the specification of ALGOL60 [Naur et al. 1963]):

$$\ulcorner \textbf{if } E=E \textbf{ then } \Gamma_0 \textbf{ else } \Gamma_1 \urcorner = \ulcorner \Gamma_0 \urcorner$$

for any well-defined expression E. However if E causes a side effect Γ', the result is at best $\ulcorner \Gamma';\Gamma_0 \urcorner$ but possibly even $\ulcorner \Gamma';\Gamma_1 \urcorner$ since the second evaluation may give a different result since it depends on the state.

We incorporate side-effects into our theory by making the value of $\mathcal{E}[\![E]\!]\rho\sigma$ a *pair* consisting of a value and a new state and define a new *application* operator

$$*: [S \to [V \to [V \times S]] \times [S \to [V \times S]] \to [S \to [V \times S]]$$

so that we have a form of *β-reduction* which takes account of side-effects:

$$\mathcal{E}[\![(\lambda I.E)E_1]\!]\rho = (\mathcal{E}[\![\lambda I.E]\!]\rho) * (\mathcal{E}[\![E_1]\!]\rho).$$

Our new interpretation function has type

$$\mathcal{E}: \mathrm{Exp} \to [[L \to V] \to [S \to [V \times S]]].$$

A word should be said here for the distinction between *environment* and *state*. The former gives the meanings of the currently-free variables; it can be manipulated at compile-time and is capable of being saved and recalled. The state, on the other hand, cannot be restored to a former value, and is only available at run-time. It gives, amongst other things, the values at given locations in the store (and is hence changed by assignment) whereas the environment gives the location corresponding to a given identifier.

Now we introduce functions and routines. Here for the first time our value domains become explicitly recursive (including functions on itself) and hence we make actual use of the fixed-point machinery for constructing domains.

The new syntax

<Exp> ::= **function** I . E | $E_0(E_1)$ | **routine** <Cmd>

<Cmd> ::= ... **call** <exp>

gives (as an element of the value domain V) the function $I \mapsto E$; applies the (function-valued) expression E_0 to the expression E_1 ; gives (as an element of V) the command Γ and executes the (command valued) expression E.

Assuming *call by value*, we now have results like

$$\ulcorner(\textbf{function } I.E_0)(E_1)\urcorner = \ulcorner(\lambda I.E_0)(E_1)\urcorner$$

and

$$\ulcorner \textbf{call routine } \Gamma\urcorner = \ulcorner\Gamma\urcorner.$$

Of course we also have routine-valued functions as elements: $\ulcorner\textbf{function } I.\textbf{routine } \Gamma\urcorner$ and so on.

Next we provide for the *execution* of a command Γ to affect the *evaluation* of an expression E, thus:

$$\langle Exp\rangle ::= \textbf{valueof } (\Gamma \textbf{ resultis } E)$$

whose interpretation is simply the composite

$$\mathcal{E}[\![\textbf{valueof}(\Gamma \textbf{ resultis } E)]\!]\rho\sigma = \mathcal{E}[\![E]\!]\rho(\mathcal{C}[\![\Gamma]\!]\rho\sigma).$$

We have omitted to provide for recursively defined functions and routines. This is usually done by abuse of scope and assignment, but let us do it properly:

$$\langle Exp\rangle ::= \textbf{recursive } I_0 \textbf{ function } I_1.E \mid \textbf{recursive } I_0 \textbf{ routine } \Gamma$$

$$\langle Cmd\rangle ::= \textbf{recursive } I_0 \textbf{ procedure } \Gamma$$

where the last is synonymous with **call recursive** I_0 **routine** Γ. Of course the interpretation of these is the solution of the obvious fixed-point equations.

The foregoing theory is (relatively) clean and straightforward, but the introduction of jumps and (later) assignments makes it significantly more complex We shall deal first with a restricted case of jumps which Stoy [244] calls *hops*.[6]

Suppose we add the command construct

$$\begin{aligned}\langle Cmd\rangle ::= \textbf{begin } & I_0 : \Gamma_0;\ \textbf{goto } E_0;\\ & I_1 : \Gamma_1;\ \textbf{goto } E_1;\\ & \ldots\\ & I_{n-1} : \Gamma_{n-1};\ \textbf{goto } E_{n-1};\\ & I_n : \Gamma_n;\ \textbf{end}\end{aligned}$$

so that we have blocks of code beginning with a label and (except the last) ending with a jump to another label in the *same* block. Then defining a **case** statement in the obvious way using multiple conditionals, this is the same as

$$\begin{aligned}\textbf{call } (\textbf{recursive } F \textbf{ function } I.\textbf{routine case } I \textbf{ of}&\\ \textbf{begin } I_0 : \Gamma_0; \textbf{call } F(E_0);&\\ \ldots&\\ I_{n-1} : \Gamma_{n-1}; \textbf{call } F(E_{n-1});&\\ I_n : \Gamma_n;\ \textbf{end })(I_0)&\end{aligned}$$

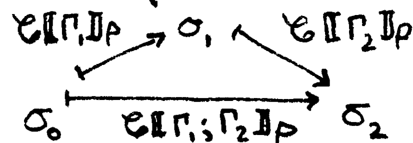
For which we perhaps ought to have the syntax

$$\textbf{recursive } P \textbf{ with } I = I_0 \textbf{ procedure } \Gamma.$$

Anyway, the *hop* essentially works by means of an interpreter which begins in state $\langle \sigma, I_0 \rangle$ and executes successively the statements Γ_i according to the previous label-part of the state, until label I_n is reached, rather analogously to a Markov chain. The interpretation is, not surprisingly, the solution of a fixed-point equation involving a kind of *permutation*.

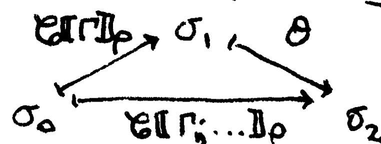
This method, which is already rather complicated, cannot deal with jumps into or out of a nested structure. Also, we remain obliged to carry through **while** loops to their conclusion, and **resultis** may only occur as the *last* statement of a **valueof** block. To provide extensions overcoming these restrictions we must employ a completely new piece of machinery called a <u>continuation</u>.

The essential problem is that the equation



provides no means for the statement Γ_1 to prevent the execution of Γ_2, except by looping or crashing, as we require if it is a jump. Because of this we abandon any attempt to interpret statements individually and seek instead an interpretation for the whole of the (remainder of the) program.

Let us assume that we know the interpretation, as a command $\theta : [S \to S]$, of the statements which would follow Γ *dynamically*, given that Γ is *not* a jump. Then that of Γ together with the following statements is



ie $\mathcal{C}[\![\Gamma]\!]\rho \circ \theta$. However if Γ is a jump, it is exactly that of the statements following the label to which it evaluates (we identify the label with these statements), and is independent of θ.

Thus θ must now be included as an argument of $\mathcal{C}$, which now has functionality (taking account now of the distinction between identifiers and locations)

$$\mathcal{C} : \mathrm{Cmd} \to [[\mathrm{Ide} \to V] \to [[S \to S] \to [S \to S]]]$$

The equations for $\mathcal{C}$ are in Stoy [258-9]. Since expressions have side-effects possibly including jumps, they also must have

continuations, but we shall neglect the details of these.

Unfortunately, we are obliged to calculate θ *backwards* from the (*dynamic*) end of the program, and where loops and more complex (eg structure-breaking) jumps occur, fixed-point methods are explicitly required. In addition to **goto** statements, we may implement **break** and **return** to exit from **while** loops and **routines** prematurely. We have the equation

$$\begin{array}{ccccc} & \mathcal{C}[\![\Gamma_1]\!]\rho \swarrow & \theta_1 & \nwarrow \mathcal{C}[\![\Gamma_2]\!]\rho & \\ \theta_0 & \longleftarrow & \mathcal{C}[\![\Gamma_1;\Gamma_2]\!]\rho & \longleftarrow & \theta_2 \end{array}$$

in place of the troublesome one before: note the reversal of the arrows.

We shall now turn to assignments, the only commands we shall meet which *explicitly* change the state. This is the last of the commonly occurring problems apart from self-modifying code, which was employed by very early machine-code programmers in the absence of suitably-powerful addressing modes [eg Hollingdale & Tootill 1963 p.133]. However this is such a heinous crime that it would be quite improper for us to provide succour to its perpetrators: perhaps the gospel of Wirth [ ] will eventually outlaw **goto** and its relatives as well.

The meaning of a variable on the *left* of an assignment is different from that on the *right*, being (roughly) an *address* rather than a *value*. The terms <u>$\mathcal{L}$-value</u> and <u>$\mathcal{R}$-value</u> are used in these senses; a general expression (eg including arithmetic operators) still has just an <u>$\mathcal{E}$-value</u>. Now an environment is a function mapping *identifiers* to *locations*, $\rho : [\mathrm{Ide} \to \mathrm{L}]$, which gives their *meanings*. The state includes a function $[\mathrm{L} \to \mathrm{V}]$ which gives the actual values, and also a *flag* $[\mathrm{L} \to \mathrm{T}]$ indicating whether a location is in use; it is an *error* to attempt to look up an unused location. Thus essentially $\mathcal{L} : [\mathrm{Ide} \to \mathrm{L}]$ and $\mathcal{R} : [\mathrm{Ide} \to \mathrm{V}]$.

We now need management functions such as *Update*: $\mathrm{L} \times \mathrm{V} \to [\mathrm{S} \to \mathrm{S}]$ to make an assignment, *New*: $\mathrm{S} \to \mathrm{L}$ to provide a free location, *Extend*, *Lose*: $\mathrm{L} \to [\mathrm{S} \to \mathrm{S}]$ to obtain and release it and so on. As well as the *scope* and *visibility* of a variable we must also decide on its *extent*, ie how long a physical location remains allocated to it, which in the presence of structure-

breaking jumps is a non-trivial problem.

In terms of applications, we are still a very long way from describing the behaviour of a compiler. For this purpose we provide another semantic theory more resembling an actual machine. This is called *stack semantics*. The advantage of carrying this out *formally* is that we then have some prospect of (hopefully automatically) *proving* the correctness of a given compiler as a *model* of the specified language.

More generally, with the escalation of the size, complexity and hence cost of software, it is becoming more important to provide sophisticated techniques of program-proving. The verification of a typical computer program is a more difficult problem than that of a mathematical proof at a similar level of detail, because the former lacks the *referential transparency* of the other. If one omits a page of a piece of mathematics, one may fail to understand the proof, but if one omits several lines from a program it may become *incorrect*, because an assignment *dynamically changes the meaning* of a variable. For this reason much work is now being put into the definition of *non-sequential* and *functional* languages[7], whose programs consist of *axiomatic* rather than *imperative* statements.

Another technique is to decorate each block of code with conditions at the beginning and the end, the latter saying what can be deduced after successful completion of the block given that the former was true before it was begun. Of course the existence of jumps into such a block would invalidate the technique.

Finally, let us review the lattice theory which we have used in the discussion of semantics. We have certainly made use of *bottom* and *directed sups* in order to provide for *undefined* and recursion. We have not used *infs* or *binary sups* and have only hinted at possible meanings for *top*. In fact binary sups will become important in the next chapter because they represent *cooperation*, but an argument can be made for abolishing top altogether.

As we remarked in the first chapter, the points in the lattice of our data-type may be viewed as *statements* which may be made about an element. The element itself is then identified with the collection of all its properties [Scott 1980], which is an ideal, I, of the lattice, being closed under approximation and conjunction (ie $x \le y \in I \Rightarrow x \in I$ and $x, y \in I \Rightarrow x \vee y \in I$). A *total* element is then a *maximal* (proper) ideal; since the algebra of statements is usually written with the opposite order this is usually called an *ultrafilter*. Top then represents that element about which *everything* is true, which in logic is regarded not as *perfect* information but as *contradiction*.

Are we then justified in saying that top represents *error*? A better interpretation would be *crash* (which goes nicely with *looping* for bottom) since when dealing with top we usually require it to be preserved (this is the case, for instance, with Lawson-continuous maps). However programs which crash on the least error are *not* very popular, and error-recovery is usually at least *attempted* (this is one case where structure-breaking jumps may be justified). Thus error-elements should be provided as extra points in our domains which are capable of continuous and non-destructive testing: see Stoy [114-5].

Thus top plays no useful rôle, and in most discussions it requires to be treated as a special case. More seriously, it actually gets in the way of the powerdomain construction which is the subject of the next chapter.

CHAPTER VI POWERDOMAINS AND NONDETERMINISM

In this chapter we shall consider how the theory is extended to cover parallel processing and other nondeterministic features. For this we shall need to construct a powerdomain of subsets of a given domain of computation states, and this will be amongst the functions in domain equations for certain semantic theories. Unfortunately, top is now not merely *inconvenient*, it actually becomes a nuisance, so our domains shall henceforth be assumed to be algebraic cpos. We shall show that the appropriate objects to consider are *profinite* cpos.

In chapters I and II we discussed two particular domains (continuous lattices), namely $D_{\mathbb{R}}$ and $P\omega$. In their respective natural representations as sets of sets of numbers (reals and naturals) they were contrasted in that *larger* sets were considered respectively *worse* and *better*. Elements of $P\omega$ were meant to give an approximate picture of, say, a function; the larger the element, the better the picture. Indeed the order on $P\omega$ corresponds to the ordering of partial functions by *ignorance*. With the reals, on the other hand, the object of a calculation is to *cut down* the interval within which the desired result lies.

In general with distributive processing we may either have the parts *cooperating* to tell us *more* about the result (say by providing values of a function at many places) or else use them to provide (hopefully consistent) information about a *single* datum. In the extended version of his [1982] paper, Scott applies his information system formulation of the domain concept to these two distinct interpretations. That formulation will not be discussed here, and we shall concern ourselves with the *cooperative* interpretation; the source is Plotkin [1976].

First consider a language with a *nondeterministic branch*:

<Cmd> ::= <Ide> **:=** <Exp> | <Cmd>;<Cmd> | **if** <Exp> **then** <Cmd> **else** <Cmd> | **while** <Exp> **do** <Cmd> | <Cmd> **or** <Cmd>.

Now let

$P_1 = \ulcorner x := 1 \urcorner$

$P_2 = \ulcorner x := 0$; **while** $x=0$ **do** $(x := 1$ **or** $x := 0) \urcorner$

$P_3 = \ulcorner x := 0$; $y := 0$; **while** $x=0$ **do** $(x := 1$ **or** $y := y+1) \urcorner$

A result of such a program is a pair $\langle x, y \rangle$, and the result sets of P_1, P_2 and P_3 are $\{\langle 1, y_0 \rangle\}$, $\{\perp, \langle 1, y_0 \rangle\}$ and $\{\perp\} \cup \{\langle 1, n \rangle : n \in \omega\}$. Notice that P_2 and P_3 may fail to terminate if the second option is always taken: hence the $\perp$. It is clear that P_1 and P_2 are distinct, and that $\perp$ plays an important role.

Assuming for simplicity that the state is given by the contents of a particular location, x, which may be any value in some given domain S, we may in general consider the interpretation of a program to be a function $S \to \mathcal{P}(S)$ from the initial state to the set of possible final states. $\perp$ denotes lack of termination: the above example shows that it must not be neglected.

Suppose S is a flat domain $X_\perp$, ie whose elements are $X \cup \{\perp\}$ with $x \le y$ iff $x=y$ or $x=\perp$. From any given initial state, $s \in S$, the execution paths form a finitely-branching tree (because of the finitarity of **or**) and so by König's lemma the tree is either finite or has an infinite branch. Thus $f(s)$, the set of possible final states, is either finite *and nonempty* or else contains $\perp$. We shall define $\mathcal{P}(S)$ to be the collection of such sets in this case. It is ordered by the Egli-Milner order

$$X \le_M Y \text{ iff } (\forall x \in X \; \exists y \in Y \text{ st. } x \le y) \text{ and } (\forall y \in Y \; \exists x \in X \text{ st. } x \le y).$$

Then [Plotkin 455] $\mathcal{P}(S)$ is a cpo in which every element is the limit of an increasing sequence of finite elements and the functions $\cup$ and $\{\cdot\}$ are continuous. However the operations $\cup$ (union) and $\vee$ (sup from $\le_M$) do *not* coincide. Observe that $[S \to 2]$ will *not* do since nothing represents $\{\perp\}$.

Nonterminating computations are not necessarily meaningless, such as if we have a **print** statement. In this case we no longer have a flat lattice, since the state is to include everything which has been printed so far, which is (nontrivially) ordered by curtailment. Thus we require $\mathcal{P}(S)$ for a non-flat lattice.

We may consider a nondeterministic program as a deterministic one which takes an *oracle*[1] as a parameter, ie an infinite binary string ω which is used to arbitrate at each **or** operator. A *fair* oracle is one with infinitely many occurences of each of 0, 1, so that P_2 and P_3 are no longer nonterminating. This is rather analogous to using random variables, except that we do not introduce a measure. Ω, the domain of oracles, does, however, have a topology, and we require programs to be continuous with respect to it.

A set which is the image of the set of infinite oracles under some continuous map $f: \Omega \to S$ is said to be *finitely generable*[2]. We consider these sets *pre-ordered* by a relation which must be no stronger than Egli-Milner one, but at least as strong as that induced by the monotonicity of all functions of the form $X \mapsto \bigcup\{f(x): x \in X\}$ where $f: S \to \mathcal{P}(X_\perp)$, namely

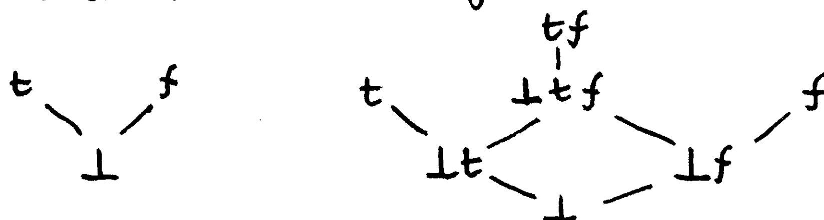
$$X \underset{\sim}{\sqsubseteq} Y \text{ iff } (\forall \text{ ctbl sets } X,\ f: S \to \mathcal{P}(X_\perp),\ \tilde{f}(X) \leq_M \tilde{f}(Y))$$

where

$$\tilde{f}(X) = \bigcup\{f(x) : x \in X\}.$$

Unfortunately [Plotkin 461] this does not determine the order on $\mathcal{F}(S)$, the set of finitely generable subsets, and we choose the weaker relation, $\underset{\sim}{\sqsubseteq}$.

The theory is somewhat unsatisfactory at this point for general domains, but we may certainly carry out the construction in the *finite* case. $\underset{\sim}{\sqsubseteq}$ induces a (nontrivial) equivalence relation on $\mathcal{P}(D)$, which is now just the powerset (but not ordered as such), and $\underset{\sim}{\sqsubseteq}$ coincides with $\leq_M$. Under the equivalence relation, each set is identified with its *convex closure*, $\mathrm{Con}(X) = \{z \in D : \exists x, y \in X \ \ x \leq z \leq y\}$, so we may as well just consider convex sets. $\mathcal{P}(D)$ then becomes a finite cpo. For example applying the construction to the domain of truth-values we get



The reason for abolishing top is now apparent: if we start with a lattice we do not necessarily get one as the powerdomain [Plotkin 463]. $\mathcal{P}$ is now rendered a functor on the category of finite partial orders by $f \mapsto \tilde{f}$ as above.

Plotkin [463-8] now introduces the category of profinite partial orders, which he calls SFP ("sequences of finite partial orders")[3] and shows, essentially, that $\mathcal{P}$ preserves cofiltered limits in it (which exist) [468-474]. For a profinite cpo, D, he characterises $\mathcal{P}(D)$ as the directed-sup completion of the set of convex closures of finite nonempty sets of compact elements of D, under the Egli-Milner order.

He then discusses several functions analogous to the set-theoretic functions of singleton and finite and infinite union, as well as cartesian product and (if the domain is a $\wedge$-semilattice) a weak form of intersection. Domain equations involving $\mathcal{P}$ may be solved by the method of chapter IV, and we shall turn to an application of this shortly.

Now that we have chosen to consider algebraic cpos rather than continuous lattices[4], we no longer have cartesian closure. This question gives another reason for choosing SFP as the appropriate category of domains. For Smyth [1982] has shown that if D and [D→D] are both algebraic cpos, then D is profinite. Thus amongst *full* subcategories of algebraic cpos SFP is the largest.

Plotkin [465-7] gives an internal characterisation of profinite cpos. Let X be a subset of a cpo D and $\mathcal{U}(X)$ the set of *minimal* upper bounds of X in D; it is *complete* if whenever u is an upper bound of X then $u \geq v$ for some $v \in \mathcal{U}(X)$. Finally let $\mathcal{U}^*(X)$ be the least set containing X and closed under $\mathcal{U}$. Then D is profinite iff it is algebraic and, for every finite set X of compact elements of D, $\mathcal{U}(X)$ is complete and $\mathcal{U}^*(X)$ is finite.

SFP (or SFP_ω) is then cartesian closed, has powerdomains and models of $\lambda\eta$. Since it has cofiltered limits and a terminal object its structure is rather similar to that of its elements.

Finally[5] let us apply this to the semantics of a language with parallel processing:

<cmd> ::= <cmd> **par** <cmd>.

This performs an arbitrary interleaving of the atomic statements of one command with those of the other.

This may be achieved, as before, with an oracle.

The meanings of statements can no longer be functions $S \to \mathcal{P}(S)$. For example [Plotkin 457], with $\Gamma_1 = \ulcorner x := 0 ; x := x+1 \urcorner$, $\Gamma_2 = \ulcorner x := 1 \urcorner$, then although Γ_1, Γ_2 have the same meaning as functions, $\ulcorner \Gamma_1$ **par** $\Gamma_2 \urcorner$ and $\ulcorner \Gamma_2$ **par** $\Gamma_2 \urcorner$ have quite different meanings. Suppose our meanings are to be elements of a domain R, of *resumptions*. The execution of Γ on a state σ either results in a final state, or in some state and point in Γ from which computation can be resumed, so either $r(s) \in S$ or $r(s) \in S \times R$. Then we want $R \cong [S \to \mathcal{P}(S + S \times R)]$.

The interpretation of the other statements is rather simpler. We dealt, essentially, with expressions and assignments in chapter V, and we define the obvious composition function to cover sequencing. This only leaves **or**, which naturally employs *union*. Thus our nondeterministic semantics is complete.

FOOTNOTES AND CONCLUDING REMARKS

Chapter I

1. I have no patience for those who claim that 3.14159 represents anything other than the *rational* number $\frac{314159}{100000}$. In particular $3.14159 = 3.141590 = 3.1415900000\ldots$
2. In this chapter I shall try to explain the mathematical terms for the benefit of the intelligent layman; there seems little point in trying to do so later. A partial order $\leq$ on a set D is a relation such that (a) $x \leq x$ for all $x \in D$, (b) $x \leq y,\ y \leq z \Rightarrow x \leq z$ for all $x, y, z \in D$ (c) $x \leq y,\ y \leq x \Rightarrow x = y$ for all $x, y \in D$. A least element $\perp$ is such that $\perp \leq x$ for all $x \in D$. A directed set $S \subseteq D$ is a nonempty subset such that $x, y \in S \Rightarrow \exists z \in S$ with $z \geq x, z \geq y$. The sup of a (directed) set $S \subseteq D$ is the element $\bigvee S$ such that $\bigvee S \geq x$ for all $x \in S$, and if $y \geq x$ for all $x \in S$ then also $y \geq \bigvee S$.
3. This seems to be standard, but [Comp. VII 3.3] uses cpo for something else, and Plotkin [1976] calls it an ipo (inductive partial order).
4. A binary sup or join is the sup of a two-element set: see note 2.
5. Monotonicity is sufficient if restricted to the *compact* elements of an *algebraic* lattice (p.10)
6. Preservation of top and error elements is required in other definitions of this term (p.32 and Stoy p. 178)
7. But we shall nevertheless be forced to assume our lattices algebraic in chapter III (p.16).
8. sups: ie of all sets (see note 2); infs: the same but with the inequalities reversed
9. Compendium or Comp. denotes throughout Gierz et al. 1980. Several other references are also assumed (in particular Scott [1976]) and two or three figure numbers refer to *pages* not *dates*.
10. Has a base of compact neighbourhoods.
11. Scott's reformulations are legion, and very difficult to relate to one another.
12. If X is a set with more than one element, there is no bijection between X and either its powerset (set of subsets) or function space (functions $X \to X$).
13. This and a later comment about Markov chains (p.29) are included to provide illustration for the mathematicians.
14. I shall use **bold** face for keywords of a typical

(unspecified) programming language.

15. Almost in a Platonic sense!
16. Apologies to the logicians for my lack of respect for constructivism!

Chapter II

1. The computer scientists seem to have failed to grasp the difference, indeed the whole point of λ-calculus.
2. An operation which mathematicians consider too trivial to be worthy of a name.
3. One might be forgiven for thinking that Cambridge CST students write dissertations on nothing else.
4. See also p.20.
5. I still remain unconvinced of the connection between lattice theory (and $P\omega$) and the λ-calculus.
6. This seems artificial and I dont believe it.
7. Hence λ = graph is left adjoint to $\bar{\varepsilon}$ = func.
8. See p.8.
9. See Manna & Shamir [1976] for an argument for *optimal* fixed points. Their example is the discrete solution of Laplace's equation: a very long way from this essay!
10. See pp. 4-5
11. See Scott [1972]

Chapter III

1. When dealing with category theory rather than λ-calculus, I prefer to put maps on the *right*.
2. **int** and **bool** are examples of <u>flat lattices</u>, in which $x \leq y \Rightarrow x = y$ or $x = \bot$ or $y = \top$.

Chapter IV

1. Maps are on the right in category theory.
2. The terms $\lambda\alpha$-, $\lambda\beta$- and $\lambda\eta$-lattice are my own. "$\lambda\alpha$" is merely a convenience : of course it is far too weak to be any kind of model of λ-calculus.
3. Write <u>CL</u> and <u>AL</u> for the categories of continuous (respectively algebraic) lattices and Lawson-continuous maps.
4. I'm a little uncertain about this.
5. This will be relevant in chapter VI.
6. My *guess* is that this will work.
7. "D_∞" seems to be more a piece of folklore than the name of a particular lattice.

8. For $(\psi,\varphi):X\to Y$, ie $\psi:X\to Y$ and $\varphi:Y\to X$, φ and ψ satisfy $\varphi\psi=1_Y$ and $\psi\varphi\leq 1_Y$ in $Hom(Y,Y)$.
9. And with good reason, for Funct preserves 0 and so ~~this~~ initial object is useless.

Chapter V.

1. I here beg the indulgence of the Part III examiners, who are invited to consider this chapter an *appendix*. However it would be a pity to build up this theory and not see how it's used.
2. Requiescat in pace.
3. ditto.
4. I'm afraid I've treated the relevant distinctions rather poorly.
5. Of course an incorrect use of the word.
6. He explains it very badly.
7. See, for example, Darlington *et al.* [1982]

Chapter VI

1. One may consider an oracle as a noncomputable function; there is a theory of enumeration degrees [Scott 1976, pp. 535-9] which discusses *how* noncomputable a function can be.
2. Plotkin, however, shows this notion to be inadequate but does not give a solution to the problem.
3. I have ignored all cardinality considerations (see pp. 5-6)
4. With due apologies to Peter Johnstone!
5. Phew!

Remarks

I am conscious in this essay of having attempted to cover several major areas of research in both Mathematics and Computer Science without being able to do justice to any of them. I have also transgressed the traditional boundaries between these two subjects, but for this I make no apology since I refuse to recognise any significance for these purely *administrative* divisions.

Having drawn the conclusion that continuous lattices (which were, after all, the raison d'être of this essay) are inappropriate for this subject — top has been dismissed as a troublemaker and *algebraic* conditions have been employed in place of *continuous*

ones – it may with some justification be argued that they have no place in the title. This would not, however, give due credit to their place in the *history* of the subject, or in its future development. For it was, after all, Scott's realisation that injective T_0 spaces (and hence continuous lattices) had suitably small function spaces that enabled models to be found *naturally* for the lambda-calculus and hence provided the impetus for a *mathematical* semantics.

As to the distinction between algebraic and continuous lattices, the choice of the former in most studies so far is largely the result of a *technical* failure in the theory as remarked in chapter III. It is clear that the *continuous* condition will have to be resurrected in some form to deal with $D_{\mathbb{R}}$.

My treatment of λ-calculus has been quite inadequate, and in particular there has been no time to go deeply in to the model theory of it or to consider the work of Martin Hyland in it. My inclination has been towards any interesting *category* theory which can be found, and it is clear that much is yet to be done in the study of Karoubian categories, filtered colimits and also SFP.

It was not really my aim to stray far into Computer Science, although formal semantics and program proving are active and interesting areas of current research with obvious applications in the new technology. It would be interesting to see a re-application of these techniques to Mathematics with a view to the comprehension of some of the more unwieldy proofs which are now becoming necessary.

All in all, I trust that those in each of the areas of which I have given a clumsy discussion will treat this survey with the sympathy due in consideration of the breadth of the field and the paucity of the time available.

BIBLIOGRAPHY

Banaschewski, B. & Hoffman, R.-E. (eds)

[1981] *Continuous Lattices.* Proc. Bremen 1979; Springer LNM 871

Barendregt, H.P.

*[1981] *The Lambda Calculus: its Syntax and Semantics.* North-Holland Studies in Logic 103

Böhm, C. (ed.)

[1975] *λ-calculus and Computer Science Theory* Proc. Rome 1975 Springer Lecture Notes in Computer Science 37

Clarke, T.W., Gladstone, P.J.S., Maclean C.D., Norman A.C.

[1980] *SKIM - the S,K,I reduction machine* in Proc. LISP-80 Conf.

Darlington, J., Henderson, P. & Turner, D.A. (eds).

[1982] *Functional Programming and its applications* CUP

Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M. & Scott, D.S. [≡ Compendium]

*[1980] *A Compendium of Continuous Lattices* Springer

Hollingdale S.H. & Tootill G.C.

[1963] *Electronic Computers* Pelican

Hyland, J.M.E.

[1975] *A Survey of some useful partial order relations on terms of the λ-Calculus* in [Böhm 1975]

[1981] *Function spaces in the category of locales* in [Banaschewski & Hoffmann 1981]

Johnstone, P.T.

[1983] *Stone Spaces* CUP

Lawvere F.W. (ed.)

[1972] *Toposes, Algebraic Geometry and Logic* Proc. Dalhousie 1972. Springer LNM 274

Mac Lane, S.

*[1971] *Categories for the working mathematician* Springer Graduate texts in Mathematics 5

Manna, Z. & Shamir A.
[1976] *The theoretical aspects of the optimal fixed point.* SIAM J. Comp. 5 (1976) 414-426

McGettrick, A.D.
[1978] *Algol 68: a First and Second Course* CUP

Naur, P. *et al.*
[1963] *Revised Report on the algorithmic language ALGOL 60* Comp. J. 5 (1963) 349-367

Plotkin, G.
*[1976] *A Powerdomain Construction* SIAM J. Comp. 5 (1976) 452-487

Scott, D.S.
[1970] *Outline of a Mathematical Theory of Computation* PRG-2
[1970a] *The Lattice of Flow Diagrams* PRG-3
[1972] *Continuous Lattices* PRG-7 & in [Lawvere 1972]
*[1976] *Data Types as Lattices* PRG-5 & SIAM J. Comp. 5 (1976) 522-587
[1980] *Lectures on a Mathematical Theory of Computation* PRG-19
[1982] *Domains for Denotational Semantics* in Proc. ICALP Aarhus, Denmark. Springer

Scott, D.S. & Strachey C.
[1971] *Towards a Mathematical Semantics for Computer Languages* PRG-6

Smyth, M.B.
[1982] *The Largest Cartesian closed category of Domains* CSR-108-82

Smyth, M.B. & Plotkin, G.
[1978] *The category-theoretic solution of recursive domain equations* DAI research rep. no. 60, Edinburgh.
[1982] *The Category-theoretic solution of recursive domain equations* CSR-102-82

Stoy, J.E.
*[1977] *Denotational Semantics – the Scott-Strachey approach to Programming Language theory.* MIT Press.

Strachey, C.
[1973] *The varieties of programming language* PRG-10

Strachey, C. & Wadsworth, C.P.
[1974] *Continuations - a Mathematical Semantics for handling full jumps* PRG-11

Wadsworth, C.P.
[1976] *The Relation between Computational and Denotational properties for Scott's D_∞ models of the λ-calculus*
SIAM J.Comp. 5 (1976) 488-521

Notes.
CSR-n-y denotes the nth internal research report of the Department of Computer Science, University of Edinburgh.
LNM denotes Lecture Notes in Mathematics
PRG-n denotes the nth technical monograph of the Programming Research Group, University of Oxford.
* indicates that frequent reference to this is made without quoting the date.

ACKNOWLEDGEMENTS

To Martin Hyland, Peter Johnstone and Andy Pitts in DPMMS for much valuable and patient advice and general tolerance to me over the past year.

To Arthur Norman and Mike Gordon in the Computer Laboratory in Cambridge for general advice and for pointing me towards Edinburgh.

To Alan Mycroft, Mike Smyth and others in the Computer Science department in Edinburgh for inviting me up there to discuss it all.

To Joseph Stoy in Oxford for sending me some useful papers.

To all the poor fools who asked me what my Part III essay was about and stuck out my explanation, thereby making me think about it.

To the University Library catalogue room for being such a dreadful place that I couldn't face looking up another reference and thereby make this essay even longer.

To Buckinghamshire Education Authority for paying me even fewer peanuts than they did three years ago: a much more significant part of my income over the past year has come from Shape Data Ltd. in Cambridge (writing FORTRAN!)

Comments on "... Continuous Lattices ..." by Paul Taylor

I enjoyed reading this essay, and learned a good deal from it — which is more than one can usually say about Part III essays. I'm not, of course, competent to comment usefully on the computer-science aspects of the essay; my comments on the mathematical aspects are mostly of the carping variety, but shouldn't be allowed to detract too much from the overall impression that this is a fine piece of work.

Page: If you really mean $D_{\mathbb{R}} \cong \Omega(\mathbb{R})$, then its top element isn't compact; perhaps you mean $D_{\mathbb{R}}$ to be the $\wedge$-semilattice of cocompact elements of $\Omega(\mathbb{R})$ (i.e. bounded closed sets in $\mathbb{R}$) with $\perp$ adjoined.

Page 18: $K(M)$ is only "pseudo-universal"; i.e. if $F: M \to \mathscr{A}$ where $\mathscr{A}$ has split idempotents, the extension $\bar{F}: K(M) \to \mathscr{A}$ is unique only up to unique natural isomorphism (unless — which rarely occurs — every idempotent in $\mathscr{A}$ has a unique splitting). In particular, this means that K doesn't preserve "honest" colimits but sends them to pseudo-colimits; consider the coequalizer of

$$(\mathbb{N}, +) \underset{n \mapsto 2n}{\overset{id}{\rightrightarrows}} (\mathbb{N}, +).$$

Page 20: $\varphi: E \to D$ preserves all infs & directed sups iff it is Lawson-continuous and a $\wedge$-semilattice homomorphism (Stone Spaces, VII 3.3). For example, the join map $\vee: [0,1] \times [0,1] \to [0,1]$ is Lawson-continuous but doesn't preserve binary $\wedge$.

Page 36: Not having Plotkin's paper at hand, I'd have liked to know more about the category SFP. What are its morphisms? In particular, is it the dual of the category of distributive lattices (cf Stone Spaces VI 3.3)? If so, then the functor $\mathcal{P}$ should have an interesting interpretation as a finitary functor on distributive lattices — does it have anything to do with the functor V_f introduced in Lemma 1.8 of the enclosed paper?

Peter Johnstone