1. THE DEFINITION OF FUNCTIONS

It has been pointed out by Strachey [7] that many mathematicians treat functions as 'second class objects', denying them the full generality which is accorded to variables. This attitude is found right at the beginning of algebraic teaching. For example, in an expression like $(x+y)\times(x-y)$ it is understood that x and y may vary but, of course, +, × and - may not. Later, in the notation $f(x)$, the student tends to think of x as the variable, and f as a 'constant' function. Similarly, in ALGOL we can write

(if x > 1 then a else b) + 6

but not

(if x > 1 then *Sin* else *Cos*)(x).

Now when we come to consider the problem of defining functions in all their generality it is necessary to be rid of these mental restrictions. We shall mark our emancipation by writing fx instead of $f(x)$, and understand that f is some element drawn from a set of functions and x from a set of operands; or better still, that they are drawn from the same set since there is then no restriction on using functions as arguments of functions. The multiplicity of operators (addition, multiplication, etc.) is replaced by one, *application*, which is denoted by the juxtaposition of symbols in fx. If a = *Sum*, b = *Difference* and c = *Product*, then $c(axy)(bxy)$ denotes $(x+y)\times(x-y)$ - but it may equally well denote many other expressions when a,b and c are different functions.

Having said this much, we must add a few qualifying comments.

(1) It is agreed that application associate from the left; thus abc denotes $(ab)c$.

(2) From time to time we shall revert to the conventional notations, in the interests of readability, when no ambiguity is likely.

(3) Using a single domain for functions and arguments leads to a great simplification of our theory, but it also gives rise to the fundamental problem referred to in the introduction. It might seem better to put up with the added complexity of multiple domains in

order to avoid the problem. However, Scott [5] has shown that the same problem is inherent in the idea of stored program, which is clearly not a thing we can dispense with.

Suppose S is a domain of the type we have in mind: if $a \in S$ then $a : S \to S$. Using functional application we may write down expressions involving constants from S or variables ranging over S, but these expressions are not in themselves functions. This can be seen by an example: if we regard $x+1$ as the successor function, we are soon enmeshed in ambiguities. Is $x+c$ a function of one or two variables, and if the latter, in which order are they applied? To define the successor function unambiguously we need to write something like

f, where $fx = x+1$ for all non-negative integral x.

Similarly in our domain S we may define the function f corresponding to the expression $....x....$ by writing

f, where $\forall\, x \in S,\ fx = ...x....$.

This serves the purpose, but it has the disadvantage that we cannot define a function without giving it a name. It turns out that we often want to use a function in an expression without needing to name it, so that this method of definition leads to unnecessarily complex notation.

Church [2] avoided this difficulty by introducing a second operation of 'abstraction' which is complementary to application and which he represented by the lambda-notation. This specifies a function in the following manner:

1.1 Definition. Let M be an expression which takes values in the domain D. Then the function $f : D' \to D$ given by

$$\forall\, x \in D',\ fx = M$$

is denoted by $\lambda x{:}D'.M$. If the domain D' is understood from the context this may be simply written $\lambda x.M$. The operation of forming a function from an expression in this way is called *abstraction*.

In the example used previously we noticed that the successor function could be represented by f in $\forall\, x \in D, fx = x + 1$ where D is here the set of non-negative integers. Thus the lambda-notation

allows us to write it as $\lambda x{:}D.x + 1$, or simply as $\lambda x.x + 1$ if D is understood. Hence we have $(\lambda x.x+1)0 = 1$, $(\lambda x.x+1)1 = 2$ etc.

The last observation can clearly be generalized. Let N be an expression taking values in D' (and enclosed in parentheses, if necessary for what follows). Then $(\lambda x.M)N$ is equivalent by 1.1 to fN where $\forall\ x \in D', fx = M$. Thus we may evaluate $(\lambda x.M)N$ by substituting N for every occurrence of x in M.

When the lambda-calculus is set up as a formal mathematical system this is given the force of an axiom (called the β-Conversion Rule). Since our point of view is different, being concerned with the problem of defining functions, we regard it as a consequence of 1.1. However, we will also give the usual formulation later, together with a precise definition of 'substitution'.

So far we have been using notation rather loosely. We must now define the well formed formulae (wff) of the lambda-calculus rigorously.

1.2 Definition. A wff is either

(1) a variable;
(2) the application MN of two wff M and N (with application associating from the left);
(3) the abstraction $\lambda x{:}D.M$ (or $\lambda x.M$) of a wff M, where x is a variable; or
(4) (M), where M is a wff.

The point (.) used in abstraction is a type of bracket. When $\lambda x.M$ occurs in a larger expression, M is taken as extending either to the first unmatched closing parenthesis or to the end of the expression, whichever is first. This means that in a case like $(\lambda x.\lambda y.N)ab$ we may insert parentheses to give $(\lambda x.(\lambda y.N))ab$, and then, using association, $((\lambda x.(\lambda y.N))a)b$. Thus in the evaluation a is first substituted for x in $\lambda y.N$; b is then substituted for y in the result.

There are two axioms expressing important properties of application and abstraction.

1.3. Axiom of Extensionality. If, for some $f,g \in D$, $fx = gx$ for all $x \in D$, then $f = g$.

1.4. Axiom of Comprehension. If M is a wff in x (including the case of a wff which is independent of x), then there is a function $f \in D$ such that $\forall\, x \in D$, $fx = M$.

The Axiom of Comprehension guarantees that the abstraction can always be formed; for every wff M, $\lambda x.M \in D$. The Axiom of Extensionality then ensures that the result of an abstraction is unique.

Before we can define substitution and give the conversion rules it is necessary to define the terms 'occurs free' and 'occurs bound'. This is done by showing how to decide whether x occurs free (bound) in (a) a variable, (b) an application, (c) an abstraction. To decide whether x occurs free (bound) in an arbitrary expression we make the decision firstly for the variables it contains and then for successively larger sub-expressions up to the expression itself.

1.5. Definition.

(1) (a) x occurs free in x (but not in $y \neq x$);
(b) x occurs free in XY iff x occurs free in either X or Y (or both);
(c) x occurs free in $\lambda y.X$ iff $x \neq y$ and x occurs free in X.

(2) (a) No variable occurs bound in an expression consisting of a single variable;
(b) x occurs bound in XY iff it occurs bound in X or Y (or both) .
(c) x occurs bound in $\lambda y.X$ iff $x = y$ or x occurs bound in X.

Take, for example, the expression $(\lambda x.ax)x$. a and x both occur free in ax. In $(\lambda x.ax)\ a$ is still free, but x occurs bound and not free. In $(\lambda x.ax)x$, x occurs both free and bound.

It was said above that $(\lambda x.M)N$ is evaluated by substituting N for x in M. However this is only true if a rather elaborate definition of 'substitution' is used, since otherwise incorrect evaluation occurs. For example, we have already seen that the first step in evaluating say $(\lambda x.\lambda y.yx)yz$ is to 'substitute' y for x in $(\lambda y.yx)$. But a simple literal substitution will give $(\lambda y.yy)$ whereas the correct result is something like $(\lambda u.uy)$ - the bound occurrence of y in $(\lambda y.yx)$ must not be confused with the free occurrence in yz so we first rewrite $(\lambda y.yx)$ as $(\lambda u.ux)$ before performing the literal substitution.

The definition of a substitution operator which achieves the desired result is given by Curry and Feys [3]; the reader will find a detailed justification in chapter 3 of their book.

1.6. Substitution Rules Let x be a variable and X and M wff. Then $[M/x]X$ is the wff X' defined as follows:

Case 1. X is a variable.

(a) If $X = x$ then $X' = M$

(b) If $X = y \neq x$ then $X' = X$

Case 2. $X = YZ$. Then $X' = Y'Z'$.

Case 3. $X = \lambda y.Y$.

(a) If $y = x$ then $X' = X$.

(b) If $y \neq x$ then $X' = \lambda z.[M/x]([z/y]Y)$

where z is defined as follows:

(i) if x does not occur free in Y
or if y does not occur free in M
then $z = y$;

(ii) if x does occur free in Y and y does occur free in M then z is the first variable in a list of variables such that $z \neq x$ and z does not occur free in either M or Y.

The crux of 1.6 is in case 3 (b)(ii). We can see how this works by considering again the evaluation of $(\lambda x.(\lambda y.yx))yz$. The result of substituting y for x in $(\lambda y.yx)$ is denoted $[y/x](\lambda y.yx)$ so that we have, for the purposes of 1.6, $M=y$, $x=x$, $X=\lambda y.yx$, $y=y$ and $Y = yx$. Case 3(b) holds, and since both x occurs free in Y and y occurs free in M alternative (ii) holds. Thus

$$\begin{aligned}[y/x](\lambda y.yx) &\rightarrow \lambda a.[y/x]([a/y]yx) \\ &\rightarrow \lambda a.[y/x](ax) \\ &\rightarrow \lambda a.ay\end{aligned}$$

which is essentially the result previously given.

In the usual formulations of the λ-calculus there are several 'conversion rules' which have the force of axioms. We have already noticed the β-rule, which we regard as deriving from 1.1. Also, we have made implicit use of α-conversion, which provides that bound variables may be changed systematically when no confusion

results, so that for example we may write $(\lambda u.ux)$ or $(\lambda a.ax)$ for $(\lambda y.yx)$ before substituting y for x. A third conversion rule is relevant here also, η-conversion. This is justified by the fact that both $\lambda x.(Mx)$ and M give My when applied to y.

1.7. Conversion Rules.

(α) If y is not free in X $\quad \lambda x.X \ \mathrm{cnv}_{\alpha} \ \lambda y[y/x]X$;

(β) $(\lambda x.M)N \ \mathrm{cnv}_{\beta} \ [N/x]M$;

(η) if x is not free in M, $\lambda x.(Mx) \mathrm{cnv}_{\eta} \ M$.

In these rules cnv indicates that either side may be replaced by the other.

In using the λ-calculus we are particularly interested in evaluating λ-expressions, which means in practice that we wish to eliminate abstractions as far as possible. Both β- and η-conversion allow us to do this by replacing an expression in the form occurring on the left of the cnv symbol by the corresponding expression on the right, as we shall see in the examples 1.8. When a conversion is applied in this way it is called a *reduction* and the expression which was converted is called a *redex*. Thus for any wff M and N

$(\lambda x.M)N$ is a β-*redex*, and

$\lambda x.(Mx)$ is an η-*redex* (x not free in M).

The notation used for reduction is

(β-*reduction*) $\quad (\lambda x.M)N \ \mathrm{red}_{\beta} \ [N/x]M$

(η-*reduction*) $\quad \lambda x.(Mx) \mathrm{red}_{\eta} \ M$.

An expression containing no redexes is said to be in *normal form*.

1.8. Examples

(1) $\quad (\lambda x.\lambda y.y)ab \ \mathrm{red}_{\beta} \ (\lambda y.y)b$

$\qquad\qquad \mathrm{red}_{\beta} \ b$

Notice that whenever c is independent of x $\lambda x.c$ is the constant function which has the value c for all values of its argument. Thus $\lambda x.\lambda y.y$ is the constant function whose value is $\lambda y.y = I$, the identity function.

(2) $\quad (\lambda f.\lambda x.f(fx))ab \ \mathrm{red}_{\beta} \ (\lambda x.a(ax))b$

$\qquad\qquad \mathrm{red}_{\beta} \ a(ab)$

(3) An expression which illustrates η-reduction:

$$\lambda x.\lambda y.axy \ \text{red}_\eta \ \lambda x.ax$$
$$\text{red}_\eta \ a$$

(4) Next consider $(\lambda x.xx)(\lambda x.xx)$. $(\lambda x.xx)$ has the form $\lambda x.(Mx)$, but is not η-reducible because x occurs free in M. An attempt at β-reduction yields

$$(\lambda x.xx)(\lambda x.xx) \ \text{red}_\beta \ (\lambda x.xx)(\lambda x.xx)$$

so that the expression has no normal form.

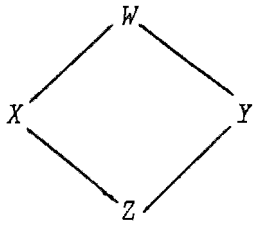
(5) $(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))b$
$\text{red}_\beta \ (\lambda y.y)b$
$\text{red}_\beta \ b.$

In all these examples we have used *normal order reduction* which proceeds by always reducing the leftmost redex. If in case (5) we had attempted instead to reduce $(\lambda x.xx)(\lambda x.xx)$ the reduction would not have terminated. This raises an important question concerning the order of reduction: since one reduction sequence may terminate and another not, is it possible for two sequences to terminate differently? Could we, for example, reduce the same expression to p by one sequence and to q by another? The conversion rules are an essential part of the evaluation 'mechanism' of the λ-calculus, and ambiguity would be serious. The answer is given by the following theorem. A proof may be found in Curry and Feys [3] chapter 4, but less arduous proofs are also known.

1.9. Theorem (Church-Rosser)

I. If X cnv Y, then there is a wff Z such that X red Z and Y red Z.

II. If A red B then there is a normal order reduction from A to B.



The uniqueness of the normal form up to α-conversion follows from 1.9, for suppose a wff W had two distinct normal forms, X and Y which were not α-convertible. Then X cnv Y (by the route $X \to W \to Y$, if by no other), so that there is a Z to which both X and Y reduce, contrary to the assumption that

X and Y are in normal form and not α-convertible. Part II guarantees that normal order reduction will always reach the normal form, if one exists.

The Church-Rosser Theorem indicates the feasibility of using the conversion rules to define an evaluation mechanism for λ-expressions. Before going into the theory further it is worth while spending a little time exploring the expressive power of the λ-calculus by looking at some practical examples.

1.10. Examples

(1) The *truth values* may be represented as follows:

$$\text{true by } \lambda x.\lambda y.x \qquad \text{and}$$
$$\text{false by } \lambda x.\lambda y.y$$

Each representation is a function; true selects the first and false the second of the two arguments which follows. To see how this works, suppose a predicate *Gr* is defined by

$$Gr\ a\ b = \lambda x.\lambda y.x \quad if\ a > b$$
$$= \lambda x.\lambda y.y \quad if\ a \leq b$$

Then

$$Gr\ x\ 0\ x\ (-x) = x \quad if\ x > 0,$$
$$= -x\ if\ x \leq 0;$$

this computes the function *Abs* normally expressed by if $x > 0$ then x else $-x$. In general, if b is a Boolean expression and b' the corresponding expression taking values in $\{\lambda x.\lambda y.x, \lambda x.\lambda y.y\}$, then $b'(e_1)(e_2)$ corresponds to if b then e_1 else e_2. For example the logical function *And* which may be written

$$\lambda u.\lambda v.\ \text{if } u \text{ then } v \text{ else false}$$

is represented by

$$\lambda u.\lambda v.u\ v(\lambda x.\lambda y.y).$$

(2) The use of the λ-notation enables us to apply a function of several variables to its variables one at a time. Take for example $Sum: R \times R \to R$ defined by $Sum\langle a,b\rangle = a + b$ (or in the more usual notation, $Sum(a,b) = a + b$). Then

$$\lambda x.Sum\langle a,x\rangle$$

is a function which adds a to its (single) argument. It is a con-

vention that we represent this function by *Sum a*, or *Sum(a)* if the notation $f(x)$ is being used. (This convention is called *Schonfinkel's Device*, and *Sum(a)* is a 'curried function' in honour of H. B. Curry.) Thus we have, in the two notations,

$$Sum\ a\ b = Sum\langle a,b\rangle = a + b \qquad \text{and}$$
$$Sum(a)(b) = Sum(a,b) = a + b.$$

(3) The combinators of Curry and Feys.
Irked by the untidiness of the process of substituting for bound variables in λ-expressions and elsewhere, Curry experimented with the expedient of eliminating variables altogether. The main combinators in the system of Curry and Feys [3] may be defined in the λ-notation.

$$\mathsf{I} = \lambda x.x$$
$$\mathsf{K} = \lambda x.\lambda y.x$$
$$\mathsf{S} = \lambda x.\lambda y.\lambda z.xz(yz)$$
$$\mathsf{B} = \lambda x.\lambda y.\lambda z.x(yz)$$

Notice that

$$\mathsf{KI} = (\lambda x.\lambda y.x)(\lambda x.x)$$
$$\mathsf{cnv}_\beta\ \lambda y.\ \lambda x.x$$
$$\mathsf{cnv}_\alpha\ \lambda x.\lambda y.y$$

so that K and KI correspond respectively to true and false in the representation which was used earlier.

(4) Pairing functions In 2.38 we shall define three functions, P, M_0 and M_1, associated with ordered pairs by
$P(x_0)(x_1) = \langle x_0,x_1\rangle, M_0(\langle x_0,x_1\rangle) = x_0$ and $M_1(\langle x_0,x_1\rangle) = x_1$.

Suppose we represent the pair $\langle x_0,x_1\rangle$ by $\lambda u.u\ x_0x_1$. This is a function which yields x_0 and x_1 respectively when applied to the representations of true and false used in 1.10(1). For example,

$$(\lambda u.u\ x_0x_1)\mathsf{K}\ \mathsf{red}_\beta\ \mathsf{K}x_0x_1$$
$$\mathsf{red}_\beta\ x_0$$

Thus M_0 and M_1 are represented by $\lambda z.z\mathsf{K}$ and $\lambda z.z(\mathsf{KI})$ respectively. P itself is represented by $\lambda a.\lambda b.(\lambda u.uab)$.

(5) Representation of the non-negative integers. Suppose we represent 0 by

$$\overline{0} = \lambda f.\lambda x.x$$

and the successor function by

$$Suc = \lambda k.\lambda f.\lambda x.f(kfx).$$

Then

$$\overline{1} = Suc\ \overline{0} = (\lambda k.\lambda f.\lambda x.f(kfx))(\lambda f.\lambda x.x)$$
$$\mathsf{cnv}_\beta\ \lambda f.\lambda x.fx$$

and

$$\overline{2} = Suc\ \overline{1}\ \mathsf{cnv}_\beta\ \lambda f.\lambda x.f(fx)$$

and so on. (We have already met $\overline{2}$ in 1.8(2).) In general, if n is an integer, the representation $\overline{n}$ has the property that $\overline{n}fx = f^n x$ - in other words, $\overline{n}$ causes its first argument to be applied n times to its second.

From what has just been said, it follows that

$$\overline{m}f(\overline{n}fx) = f^m(f^n x)$$
$$= f^{m+n}x$$
$$= \overline{m+n}\ fx$$

so that

$$\overline{m+n} = \lambda f.\lambda x.\overline{m}f(\overline{n}fx),$$

and hence we may represent the addition function by

$$Sum = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx).$$

Products and powers may easily be defined using similar methods. To define the equality predicate takes a little longer.

We first define

$$F = \lambda x.\lambda u.u(Suc(x\mathsf{K}))(x\mathsf{K}).$$

This has the property that $F\langle\overline{m},\overline{n}\rangle = \langle\overline{m+1},\overline{m}\rangle$, where ordered pairs are represented as in (4). Thus $\overline{k}F\langle\overline{0},\overline{0}\rangle = F^k\langle\overline{0},\overline{0}\rangle$

$$= \langle\overline{k},Pred\ \overline{k}\rangle \quad . \ . \ . \ . \quad (\alpha)$$

where

$$Pred\ \overline{k} = \overline{k-1} \text{ if } k > 0$$
$$= \overline{0} \quad \text{if } k = 0.$$

Using (α), we may now define *Pred* itself by

$$Pred = \lambda k.(kF\langle\overline{0},\overline{0}\rangle)(KI).$$

The reader should verify that the predicate *Iszero*, which maps $\overline{0}$ into K and every other numeral into KI, is given by

$$Iszero = \lambda k.k(K(KI))(K).$$

Next we notice that

$$\overline{m}\ Pred\ \overline{n} = \overline{n-m} \text{ if } n \geq m$$
$$= \overline{0} \text{ otherwise,}$$

so that $Iszero(\overline{m}\ Pred\ \overline{n})$ has the value K iff $n \leq m$. Hence the equality predicate is given by

$$Equal = \lambda m.\lambda n.Iszero(m\ Pred\ n) \wedge Iszero(n\ Pred\ m)$$

where ∧ is the infix notation for the logical function *And* defined at the end of example (1).

(6) Recursive definition of functions. Consider the definition of the factorial function,

$$Fact = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times Fact(n-1)$$

or in the notation of example (5),

$$Fact = \lambda n.Iszero\ n\ \overline{1}\ (Prod\ n(Fact(Pred\ n)))$$

The definitions of (5) included functions defined elsewhere; for example *Pred* involved *F* and *F* involved *Suc*. It would have been a simple matter to eliminate these functions by substitution, although the resulting expressions might be unwieldy. In the definition of *Fact* we can substitute successfully for the occurrence of *Fact* on the right only if we are evaluating an expression like *Fact* 2. In such a case the substitution process terminates (and this is our justification for regarding the formula as a definition of *Fact*); if we try to eliminate *Fact* from the right-hand side in the general case of *Fact n* we fail. Thus the definition, in the form given, is not within the scope of the λ-calculus.

Put

$$g = \lambda f.(\lambda n.\mathit{Iszero}\ n\ \overline{1}(\mathit{Prod}\ n(f(\mathit{Pred}\ n)))).$$

Then $\mathit{Fact} = g\ \mathit{Fact}$, so that Fact is a fixed point of g. Now we can define a function, Y_λ, which maps a function into one of its fixed points. Let

$$Y_\lambda = \lambda a.(\lambda y.a(yy))(\lambda y.a(yy))$$

If we put $f = Y_\lambda g$ we get, by β-conversion,

$$\begin{aligned} f &= (\lambda y.g(yy))(\lambda y.g(yy)) \\ &\ \mathsf{cnv}_\beta\ g((\lambda y.g(yy))(\lambda y.g(yy))) \\ &= gf \end{aligned}$$

so that Y_λ does indeed map g into one of its fixed points, and the use of Y_λ enables us to reduce recursive definitions to non-recursive form. In the example we have been considering, we get

$$\mathit{Fact} = Y_\lambda(\lambda f.\lambda n.\mathit{Iszero}\ n\ \overline{1}(\mathit{Prod}\ n(f(\mathit{Pred}\ n)))).$$

Example 1.10(6) leaves some loose ends. Thus if g has more than one fixed point, it is not clear that $Y_\lambda g$ computes the particular fixed point we intended to define by the recursion $f = gf$; we shall return to this question later. But a more fundamental problem is highlighted by the fact that Y_λ involves the self-application of y in yy. For suppose we define u by

$$u = \lambda y.\ \mathsf{if}\ yy = a\ \mathsf{then}\ b\ \mathsf{else}\ a$$

where a and b are distinct elements of S. Then an attempt to evaluate uu by β-conversion gives

$$uu = \mathsf{if}\ uu = a\ \mathsf{then}\ b\ \mathsf{else}\ a$$

which is a contradiction. Yet if we assume the existence of S with the property that $a \in S$ implies $a : S \to S$, then there seems to be no good reason for prohibiting the formation of u.

The same problem can be viewed from the standpoint of the cardinality of S. If S is any domain containing more than one element, then the function space S^S comprising *all* functions $f : S \to S$ has a higher cardinality than S; thus it is not possible

for our space S to be identified with the whole function space S^S.

The statement of the problem in terms of cardinality suggests a strategy for its solution. The great majority of functions in the space S^S are not merely non-computable, they are completely irrelevant for any but the most abstract considerations. Thus we must fix our attention on a subspace of S^S having the same cardinality as S, but containing only 'interesting' functions. Needless to say, this is a great oversimplification of a line of development which will occupy us from now on.

REFERENCES

[1] G. Birkhoff, *Lattice Theory*, American Mathematical Society Colloquium Publications, vol.25, Third (new) edition (1967).

[2] A. Church, *The Calculi of Lambda-Conversion*, Annals of Mathematical Studies 6, Princeton, 1951.

[3] H. B. Curry and R. Feys, *Combinatory Logic*, vol.1, North-Holland, Amsterdam (1968).

[4] H. Gericke, *Lattice Theory*, Harrap (1966).

[5] D. Scott, *Outline of a Mathematical Theory of Computation*, Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems (1970), pp.169-176.

[6] D. Scott, *Lattice-Theoretic Models for the λ-Calculus*, (unpublished)

[7] C. Strachey, *Fundamental Concepts in Programming Languages* (unpublished).

The reader who wishes to go further into either the theory or application of reflexive domains may consult the following papers by Scott.

The Lattice of Flow Diagrams, Semantics of Algorithmic Languages, Springer Lecture Notes in Mathematics, vol.188 (1971).

Lattice Theory, Data Types and Semantics, New York Symposia in Areas of Current Interest in Computer Science (Randall Rustin ed.) (1972).

Continuous Lattices, Proceedings of the 1971 Dalhousie Conference, Springer Lecture Notes.

Lattice-Theoretic Models for Various Type-free Calculi, Proceedings of the IVth International Congress for Logic, Methodology, and the Philosophy of Science, Bucharest (1972).