[2] P. McCullagh and J. A. Nelder, Generalized Linear Models, 2nd ed. Chapman and Hall: London, 1990.

A. Kinsella,
Department of Mathematics, Statistics and Computer Science
Dublin Institute of Technology,
Kevin Street,
Dublin 8.

SOME MATHEMATICAL ASPECTS OF INFORMATION TECHNOLOGY: FIXED POINTS AND THE FORMAL SEMANTICS OF PROGRAMMING LANGUAGES

Anthony Karel Seda

1. Introduction

"It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance." John McCarthy[1], 1967.

In describing Information Technology, the Web page of the recently formed Information Technology Centre at University College, Galway says this: "During the past decade Information Technology (IT) has transformed business life, from the boardroom to the shopfloor. As we generally understand it, Information Technology is an outgrowth from the computer, microelectronics, and telecommunications industries, and now comprises: computer processors and data storage devices, telecommunications, software, microprocessors, automation technologies and user interface media."

Generally speaking, users of IT need not be expert in, nor even familiar with, the technologies which support it. If this is true, it is even more true that these same users need have no knowledge of the *theory* which supports the technologies which support IT. Nevertheless, the issue of the theories underlying IT and, in particular, which areas of mathematics are important in

[1]Inventor of the programming language Lisp and pioneer of AI.

it, is itself important and interesting. At the moment, the vehicle moving all the activity in IT is the electronic digital computer, and this state of affairs is likely to persist for some time into the future. Therefore, questions concerning the relationship of mathematics to IT are often really questions concerning some more or less theoretical issue in computer science, and indeed such issues are raised in this Bulletin from time to time, sometimes in an educational context, see [16], for example. So, just which areas of mathematics are currently of importance in research and teaching in theoretical computer science and IT, and which of these areas will prove to be of enduring importance in this context? But before addressing this question, it will be helpful to say a few words about the recent history of IT.

Much of the recent and ongoing work in IT has as its focus new generation computing and is the direct result of the efforts of the Alvey and ESPRIT programmes in Europe, ICOT in Japan and the consortium known as the Microelectronics and Computer Technology Corporation in the U.S.A. Indeed, all this was directly inspired by ICOT's announcement in 1982 of its intention to build the so called fifth generation computer, prompting a global race from about 1985 onwards to build such machinery. It was found necessary within these projects, see [1, 2], to broadly divide the whole of IT into, initially, four *enabling technologies*: VLSI (Very Large Scale Integration, which is concerned with chip fabrication and computer architecture); MMI (Man-Machine Interface, or human factors in computing); SE (Software Engineering, which is concerned with putting the production of software on a scientific basis (in particular, the development and use of formal methods of verification in the manufacture of software and hardware)); IKBS (Intelligent Knowledge Based Systems, i.e. Artificial Intelligence (AI)). As a matter of fact, communications and networks quickly came to be seen as so important that they were taken to be the fifth enabling technology.

The classification just described is useful as a means of organizing the applications of mathematics to IT, and can help determine which are central and which are of lesser importance. As one would expect, all five of these enabling technologies use mathematics, to a greater or lesser extent, in the way that it is used in other sciences, that is, as a precise language in which to formulate problems and results and as a tool with which to solve these problems (even MMI uses mathematics in problems concerned with pattern recognition). Returning to our main question, and taking a glance at, say, the thirteen volumes which collectively make up [3, 13, 40], and which cover several thousand pages, shows that even the first part of our question is by no means easy to answer (and the volumes just cited cover mainly the mathematics relevant to SE and IKBS and say little about the other three areas). However, such a glance does make it clear that the answer "discrete mathematics" which is sometimes proposed in response to this question is only a small part of the story, at least when this term is interpreted to mean graph theory and combinatorics, as is often the case. Important as graph theory and combinatorics undoubtedly are, they do not explain, for example, the many uses of category theory and topology in connection with domain theory and the formal semantics of programming languages. Much less do they explain the many uses of mathematical logic in connection with program verification and within machine intelligence and robotics. Still less do they explain the use of real and complex analysis in the analysis of algorithms, and the use, say, of measure and integration in connection with probabilistic powerdomains on the one hand, and in connection with uncertainty in reasoning systems on the other (where fuzzy logic is also important). Indeed, one can continue in this vein citing seemingly endless applications of different branches of mathematics to various aspects of the theory of computation and IT, and some of these are indicated in the References at the end of this article. On the other hand, many others are not mentioned at all, and there is indeed an immense literature covering the various topics of which our bibliography is but a tiny fraction.

Devising a complete classification of all the areas of mathematics which are of importance in IT would be an interesting and valuable project in its own right, though time consuming and beyond the abilities of the author, and in any case is not the objective of this article. Instead, we propose to take one concept, that of *fixed point*, and attempt to relate it to two of the main areas, SE and IKBS, which were identified earlier. The notion of fixed point is, of course, of great importance within mathematics, and it turns out also to be central in the areas we intend to consider in the context of programming language semantics. There may well be applications of ideas concerning fixed points elsewhere within IT, but they will not fall within our scope. Thus, specifically, we consider the use of fixed points in relation to the problem of giving formal, machine independent meaning (a formal semantics) to computer programs. To do such is fundamental to the problem of formal verification of software, or the use of formal methods as it is known in industry, and we take up this issue for procedural programs in §2. In §3 we briefly consider basic ideas of formal systems and mathematical logic preparatory to the discussion, in §4, of the role of fixed points in computational logic (the declarative style of programming). Again, fixed points are fundamental in this area in order to both give meaning to programs and to gain deep insight into the computation process itself, necessary if advanced machine reasoning features are to be developed such as time dependent logics, the ability for machines to learn and so on. Such questions are themselves of importance of course given the extent to which computers control complex and important systems in modern society. Finally, in §5 we discuss briefly the uses of topology, some due to the author, which unite the two themes just described. Given space limitations, not to mention those of the writer, it is not possible to do much more here than touch on the main issues. Nevertheless, it is hoped to show, en route, that the simple concept of fixed point links in a coherent fashion a wealth of important ideas drawn from mathematical logic, recursive function theory, topology, category theory and abstract algebra in an effort to resolve the apparently simple question of what a program means. Indeed, far from being simply a matter of pressing keys on a computer keyboard, which is the end user's perception, IT has behind it a rich and fascinating theory which makes use of many fundamental ideas drawn from many parts of mathematics. That at any rate has been the experience of the author over the first decade of IT, a subject which by all accounts

is set to become one of the two or three dominant forces which shape the next century.

2. Fixed-point semantics of procedural programs

No matter what programming language it is written in, a program P computes a function f. By suitably coding data structures (lists, arrays etc.) and by a simple conjugacy, we can suppose without loss of generality (though not necessarily without loss of convenience) that f is defined on vectors $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in N^n$, for some natural number n, and takes values in N, where N denotes the set $\{0, 1, 2, \ldots\}$ of natural numbers together with zero. The essence, of course, of computer science is the translation of formal or informal algorithm into (high level program) code. It is therefore eminently reasonable from the point of view of the computer scientist to impose some resource bounds on the notion of *algorithm* that one adopts, see [15]. We shall not, however, do this here so that we employ this term in the manner familiar in mathematics and in particular in the usual sense of recursive function theory, see [21, 26]. This means that f is a partial recursive function (a *computable* function) and that its domain is a possibly strict subset of N^n; if the domain of f is all of N^n, then f will be called *total*. Let us therefore denote by $\mathcal{F}_n$ the collection of all partial functions from N^n to N and write dom(f) for the domain of f. Given two functions $f, g \in \mathcal{F}_n$, we write $f(\mathbf{x}) \simeq g(\mathbf{x})$ to mean that if one of $f(\mathbf{x})$ or $g(\mathbf{x})$ is defined, then both are defined and equal. With this notation, we may define the graph of f, graph(f), by graph$(f) = \{(\mathbf{x}, y); f(\mathbf{x}) \simeq y\}$, and as usual graph$(f)$ will be identified with f. This notion permits us to partially order $\mathcal{F}_n$ by: $f \leq g$ iff graph$(f) \subseteq$ graph(g), and we note the following two facts. (i) The nowhere defined function f_ϕ, whose graph is the empty set, satisfies $f_\phi \leq f$ for all $f \in \mathcal{F}_n$, and therefore is the *bottom element* of $\mathcal{F}_n$. (ii) $\mathcal{F}_n$ is an ω-complete partial order in that any chain $f_\phi \leq f_1 \leq f_2 \leq \ldots$ has a supremum $f = \bigcup_{m=1}^{\infty} f_m$, where f satisfies (and is well defined by) $f(\mathbf{x}) \simeq \mathbf{y}$ iff $f_m(\mathbf{x}) \simeq \mathbf{y}$ for some m.

The main theorem we will need in this section is the following, known as Kleene's first recursion theorem, see [11, 21, 26].

Theorem 1. *Suppose that $\Phi : \mathcal{F}_n \to \mathcal{F}_n$ is a recursive operator. Then Φ has a least fixed point h which is a computable function. Thus, there is a computable function h satisfying*
a) $\Phi(h) = h$,
b) if $\Phi(g) = g$, then $h \leq g$.
Hence, if h is total, then it is the only fixed point of Φ. ■

We will not give a formal definition of the term *recursive* used here, but the essence of the idea is that whenever $\Phi(f)(\mathbf{x})$ is defined, its value depends only on finitely many values of f and these values can be chosen in a computable fashion. Note that Φ is itself totally defined, that is, defined on all of $\mathcal{F}_n$, but of course $\Phi(f)$ may be a partial function. It will be convenient to write $\Phi(f;\mathbf{x})$ in place of $\Phi(f)(\mathbf{x})$.

We also will not give details of the proof of this theorem, other than those which we will need later on, and they are as follows. We define inductively the following chain (f_n) of elements of $\mathcal{F}_n$: $f_0 = f_\phi$ and $f_{m+1} = \Phi(f_m)$, and now let $h = \bigcup_{m=0}^{\infty} f_m$. The continuity of Φ, implied by recursiveness and defined later, shows that h is a fixed point; the construction shows it to be the least such; recursiveness of Φ is used to show that h is in fact a computable function.

To show how this theorem is used, we consider the following simple example which is an adaptation, in some of the detail, of an example to be found in [37]. It will serve to make clear the central problem under discussion and the manner of its solution.

Example 1 Consider the problem of finding the greatest common divisor, $\gcd(a,b)$, of the two positive natural numbers a and b. The usual way to do this is to apply the Euclidean algorithm and write $a = q(a,b)b + r(a,b)$ for unique choice of natural numbers $q(a,b)$ and $r(a,b)$, where the remainder $r(a,b)$ satisfies $0 \leq r(a,b) < b$. It is then noted that $\gcd(a,b) = \gcd(b, r(a,b))$, if $r(a,b) > 0$, and that the pair $(b, r(a,b))$ is "smaller" than the pair (a,b), so that repeated application of this technique is bound to terminate (in the required gcd). It will be convenient to regard r as totally defined by setting $r(a,0) = a$ for all a, and $r(0,b) = 0$ for all b. Then, with similar, suitably chosen exceptional values for q, we see that the identity $a = q(a,b)b + r(a,b)$ is satisfied for all a and b. Now, a procedural-language implementation P of the algorithm just described, say in PASCAL, will contain a program statement of the form "gcd := gcd(y,z)", and a mathematical formulation of the algorithm is given by

$$\gcd(a,b) = \begin{cases} b, & \text{if } r(a,b) = 0; \\ \gcd(b, r(a,b)), & \text{otherwise.} \end{cases}$$

The program, and the formulation just given, recursively or implicitly define gcd in terms of itself, and the question which naturally arises now is "What is the meaning or interpretation of such a definition?" From the point of view of computation, that is, from the point of view of *operational* or *procedural semantics*, the answer is simply that we are given an iterative procedure to calculate the function gcd. Such a meaning, closely related to the behaviour of P when running on a machine, is not in general satisfactory for purposes of formal verification, and a machine independent explicit definition is needed. The standard way to obtain this, in general, is to pass to an associated operator Φ and take the function which P is intended to compute to be the least fixed point of Φ. To see how this works for the problem in question, we define $\Phi : \mathcal{F}_2 \to \mathcal{F}_2$ by

$$\Phi(f;a,b) \simeq \begin{cases} b, & \text{if } r(a,b) = 0; \\ f(b, r(a,b)), & \text{otherwise.} \end{cases}$$

It is important to note that the definition of Φ is explicit i.e. does not involve recursion. Moreover, because $\Phi(f;a,b)$ depends only on the one value $f(b, r(a,b))$ of f, for any f and (a,b), it follows that Φ is a recursive operator. Applying Kleene's theorem, we obtain the least fixed point h of Φ, and h is a computable function. By reference to the synopsis of the proof of Kleene's theorem, given above, we note the following:
1) $f_1(0,b) = b$ for all b, and $f_1(a,0)$ is undefined for $a > 0$.
2) $f_2(0,b) = b$ for all b necessarily, and $f_2(a,0) = f_1(0,a) = a$ for all $a > 0$. It follows that $h(0,b) = b$ for all b and $h(a,0) = a$ for all a.

3) If h_1 is any fixed point of Φ, and thus satisfies the equation

$$h_1(a,b) \simeq \begin{cases} b, & \text{if } r(a,b) = 0; \\ h_1(b, r(a,b)), & \text{otherwise.} \end{cases}$$

then it is easy to check that $h_1(a,b)$ coincides with $\gcd(a,b)$ for positive a and b.

It follows from these observations that $h(a,b)$ coincides with $\gcd(a,b)$ for positive a and b and therefore that h is totally defined. Hence, h is the unique fixed point of Φ, and so we recover gcd as the unique fixed point of Φ provided we are willing to accept that $\gcd(a,0) = a$ for all a and $\gcd(0,b) = b$ for all b, which is reasonable, and in particular $\gcd(0,0) = 0$, which is not unreasonable.

The discussion of this example, even though a little accelerated, identifies many of the main points of the theory, and these points can be summarized as follows.

- There is a need for abstract models of computation. Usually these are ordered spaces (such as Scott domains, see [37], and indeed much of this theory has been heavily influenced by the work of Dana Scott and Gordon Plotkin, see [28, 29, 30, 36]) but sometimes are metric spaces or even quasi-metric spaces, see [35]. Such spaces should permit one to model the computation process itself perhaps by better and better (increasing) approximations to a limit or supremum, and should incorporate a certain finiteness, known as algebraicity, which we will not identify, again see [37]. At the very least, the domains chosen must permit the construction of fixed points of certain operators. Moreover, to model features of real programming languages they must be closed under the formation of products, sums, function space, power domain (to model non-determinism) and must permit the solution of so called recursive domain equations. One is therefore looking for a Cartesian closed category of domains. The (ongoing) search for such categories is a beautiful example of pure mathematics, with the satisfaction that at all times it is closely related to genuine problems in the design of advanced programming languages.
- Fixed points can be used, via fixed point induction, to verify programs and their properties, see [22, 23]. They also can force suitable choices of exceptional values and, more importantly, eliminate problems involving choice where computation rules are used in evaluating recursive definitions.

Whilst this is only the start of the theory, we can take it no further for we want to turn now to consideration of the other main strand of this article, namely, the use of mathematical logic in IT and the role of fixed points in that context.

3. Logic, computability and formal systems

Ever since the early discoveries made by the Ancient Greeks, there has been a strong interplay between mathematics and logic, leading to the specific area of *mathematical logic*. This subject is concerned with both analysing the reasoning used in mathematics and also contributing to that subject, especially to its foundations, by examining the limits to mathematical reasoning and to what is possible. In addition, mathematical logic is proving to be indispensable in the theory of computation and in IT for several reasons, including its use in formal verification of software and as a computational medium. We will not discuss the first mentioned, in this section, other than to give references to appropriate literature; the latter we will discuss in more detail in the next section. It will be convenient to assume that the reader is familiar with elementary notions concerned with syntax: formation of terms and well formed formulae (usually abbreviated to wff, whether in the singular or the plural) from an alphabet, and the corresponding first order language. We also assume that the reader is similarly familiar with elementary notions of semantics: interpretations, formal assignment of truth values to wff, models, logical consequence and validity, for details see [7, Chapter 1].

In modern terminology, what the Greeks conceived of is the concept of a *formal system* or *formal deductive system* in which, within a theory, one reasons from axioms (distinguished wff in the underlying first order language) by applying formal rules of inference to obtain new "truths" or theorems (this process being inductive of course). Roughly speaking, this concept is defined as follows, and a useful general reference is again [7, Chapter 1].

Definition 1 A *formal system* $\mathcal{S}(\mathcal{L}, \mathcal{A}, \mathcal{R})$ or just $\mathcal{S}$ consists of:

1) A first order language $\mathcal{L}$ called the *underlying first order language*, whose alphabet is chosen so that $\mathcal{L}$ is adequate to describe the theory one has in mind.
2) A distinguished set $\mathcal{A}$ of wff in $\mathcal{L}$ called the *axioms*; usually some computability restriction is imposed here such as "it is decidable which wff are axioms" (the set of axioms therefore forms what is known as a *recursive set* and the system is said to be *recursively axiomatizable*).
3) A set $\mathcal{R}$ of *rules of inference*.

Rules of inference take the following general form: $\frac{\text{Input}}{\text{Output}}$ where Input and Output are both sets of wff of a specific syntactic form. For example, one well known rule is *Modus Ponens* which has the form:

$$\begin{array}{l} A \to B \\ A \\ \hline B \end{array}$$

where A and B are *syntactic variables* i.e. vary over wff. Thus, for example, if the wff $(\forall x\, p(f(x))) \to q(g(a,b))$ and $\forall x\, p(f(x))$ are taken as Input, then the Output is $q(g(a,b))$.

Other examples of rules of inference can be found in [7, Chapter 1].

Definition 2 A *proof* in a formal system $\mathcal{S}$ consists of a finite string $A_1 A_2 \ldots A_n$ of wff A_i in $\mathcal{L}$ where each A_i is either an axiom or follows from earlier A_j by application of a rule of inference. The end term A_n in a proof is called a *theorem*. If a wff A in $\mathcal{L}$ is the end term of some proof (i.e. if A is a theorem), we say that A is *derivable* or *provable* and write $\mathcal{S} \vdash A$ or $\mathcal{A} \vdash A$.

This definition encapsulates a formalist or mechanical view of reasoning in which there is no meaning or semantics (it is purely syntactic): one keeps on mechanically applying rules of inference generating more and more proofs and therefore more and more theorems without regard to whether or not the theorems are "true". Nevertheless, certain immediate questions arise about formal systems for which a satisfactory answer requires truth values:

Correctness of rules of inference
This is easy to answer: a rule of inference is *correct* or *sound* provided that its Output is a logical consequence of its Input. For example, since a wff B is always a logical consequence of wff $A \to B$ and A, Modus Ponens is a correct rule of inference.
Completeness of a formal system Roughly speaking this is a question about the power of a formal system to prove anything which could reasonably be expected to be provable and it depends mainly on the choice of the set $\mathcal{R}$ of rules of inference. There are several styles of formal system in use and two in particular are the *Hilbert style* formal system and the *Gentzen style* formal system. These are described in detail in [7, Chapter 1] where the exact form of the rules of inference is given in order to handle substitutions and quantifiers. The main result concerning completeness for both these styles of formal system is Gödel's well known completeness theorem, where $\models$ is the symbol for logical consequence.

Theorem 2. *In a Hilbert style or Gentzen style formal system $\mathcal{S}$, a well formed formula A is derivable iff it is a logical consequence of $\mathcal{A}$. Thus, in symbols $\mathcal{A} \vdash A$ iff $\mathcal{A} \models A$.* ■

Incompleteness in formal systems
Recall that $\mathcal{A} \models A$ means that A is true in *every* model of $\mathcal{A}$. So, if a wff A is true in some models of $\mathcal{A}$ but false in others, then A cannot be provable by Theorem 1, and conversely. The main question which arises is whether or not in a given theory (in particular this question arose in relation to Peano Arithmetic $\mathcal{PA}$) there is a closed wff, or sentence, A which is true in the intended interpretation of that theory which is not provable. In the case of $\mathcal{PA}$ the intended interpretation is the expected one in which the domain is the set of natural numbers and the function symbols there are interpreted as addition and multiplication. The shocking answer to this question for $\mathcal{PA}$ that there are such wff is the content of Gödel's famous first incompleteness theorem, a simple, but useful, form of which is as follows, see [11].

Theorem 3. *Suppose $\mathcal{S}$ is any recursively axiomatized formal system for Peano Arithmetic $\mathcal{PA}$ in which every provable sentence* *is true in the intended interpretation. Then there is a sentence σ in $\mathcal{PA}$ which is true in the intended interpretation but not provable. Consequently, $\neg\sigma$ is not provable either (since $\neg\sigma$ is not true), and σ is called an undecidable sentence.* ■

This theorem together with Gödel's second incompleteness theorem (which roughly stated says that it is impossible to prove consistency of $\mathcal{PA}$ within $\mathcal{PA}$, where consistency means absence of contradictions), effectively destroyed Hilbert's plan to mechanize mathematics. This plan, of course, grew out of attempts to overcome paradoxes in the foundations of mathematics resulting from Cantor's theory of cardinals and the unrestricted use of power set operations, and from the desire for a proof of consistency of $\mathcal{PA}$ by finitary means. A good discussion of these results can be found in [34].

The basic reason for incompleteness in $\mathcal{PA}$ is the following, and it depends on concepts to do with computability. Suppose $\gamma : L \to N$ is a coding of the wff in $\mathcal{PA}$, so that γ is bijective, effectively computable and such that γ^{-1} is effectively computable, where L denotes the set of all wff in $\mathcal{PA}$ (Gödel's original coding was not actually bijective but simply injective, but it was decidable whether or not a given natural number n belonged to the image set of γ). Thus, given a wff A in L, we can effectively find its unique *code number* or *Gödel number* $\gamma(A)$; conversely, given a natural number n we can effectively find the unique wff $A = \gamma^{-1}(n)$ from which it came–the effectiveness of these operations is crucial. There are now two sets of interest here: one is the set $\mathcal{P}$ of all provable sentences in $\mathcal{PA}$ and its image $\gamma(\mathcal{P})$, and the other is the set $\mathcal{T}$ of all true sentences in $\mathcal{PA}$ and its image $\gamma(\mathcal{T})$. What Theorem 3 says is that $\mathcal{P} \subset \mathcal{T}$ and clearly this is iff $\gamma(\mathcal{P}) \subset \gamma(\mathcal{T})$. The heart of the matter is that the set $\gamma(\mathcal{P})$ *is* the image set of a computable function i.e. can be listed by a machine (such a set is called *recursively enumerable* or usually just r.e.) whilst the set $\gamma(\mathcal{T})$ is *not* listable by any machine; the two sets therefore cannot be equal. Indeed, the set $\mathcal{T}$ or rather $\gamma(\mathcal{T})$ is highly intractable and this fact is a deep issue with far reaching consequences.

These ideas concerning formal systems are of great importance in computing and in particular in connection with formal verification of software, see [23, 42]. However, the direction we want to pursue, in the next section, is the use of deduction as an actual computational medium, rather than as a tool of verification, and to investigate the use of fixed points in the corresponding theory.

4. Formal systems and computational logic

"From hardware design to the development of new programming languages and the construction of artificial intelligence programs, Logic is the major mathematical tool. Logic will perform the function in IT that calculus performs in other areas of engineering. It will provide IT with a rigorous theoretical foundation," see [2].

The desire to mechanize reasoning and the related notion of building robots can probably be traced back a very long way in history. Certainly Descartes dreamt of a calculus with which one could perform reasoning by algebraic manipulation, a dream which was to be fulfilled by George Boole in *The Laws of Thought* with respect to propositional logic, leading to the concept of Boolean Algebra and its great use in analysing logic circuits in the hands of Claude Shannon. Perhaps, too, Blaise Pascal, Charles Babbage and Ada Byron, Countess of Lovelace, were thinking beyond mere arithmetical calculation when designing their calculating machines; certainly Babbage and Byron appear to have encountered the main concern of SE: proving that a program does what is intended.

Coming forward in time to the early years of this century, we encounter Hilbert's plan, mentioned earlier, to mechanize mathematics. As already noted, this plan came to a dead halt due to Theorem 3 and related results. Nevertheless, there is a positive side to this provided by Theorem 2. Just because some formal systems such as $\mathcal{PA}$ contain some unprovable true statements does not mean that reasoning suddenly becomes worthless. Perhaps we can make do with the theorems or logical consequences of a theory rather than deal with the larger set of all the statements true in some particular interpretation, especially if we can automate the reasoning process itself.

Following the point just made, the landmark result came in 1965 with J. A. Robinson's Unification Algorithm and his Unification Theorem, see [24, 25]. Robinson showed that there is a single rule of inference, called *resolution*, which is sound and complete in that Theorem 2 holds for formal systems using resolution as their sole rule of inference, and moreover resolution turns out to be easy to automate. To see how this works, it has first to be shown that *any* closed wff can be cast into a syntactically different but equivalent form called *conjunctive normal form CNF* (*closed* here means that there are no free variables i.e. all variable symbols in the wff are existentially or universally quantified; *equivalent* means that the new form is a logical consequence of the given wff and vice versa). We assume that this is so; as a matter of fact, not only can it be done but it can be done by an algorithm and hence is an effective operation. A wff written in CNF takes the form of a universally quantified conjunction $\forall(C_1 \wedge C_2 \wedge \ldots \wedge C_n)$, where each C_i is a *clause*, thus C_i has the general form $L_1^i \vee L_2^i \vee \ldots \vee L_{m_i}^i$, wherein each L_j^i is a *literal*, that is, either an atom A_j^i (a propositional formula) or a negated atom $\neg A_j^i$; it is usual to understand the universal quantifier $\forall$ to be present and to omit writing it. The resolution rule of inference can now be explained, at least in its simplest form, as follows. Suppose given two clauses, the *parent* clauses, $L_1^1 \vee L_2^1 \vee \ldots L_n^1$ and $L_1^2 \vee L_2^2 \vee \ldots \vee L_m^2$ the first of which contains the literal L, say, and the second $\neg L$ (the literals L and $\neg L$ are said to *clash*). Reordering and letting C^1 and C^2 denote the disjunctions of the obvious respective remaining literals, we can write the two given clauses as $C^1 \vee L$ and $C^2 \vee \neg L$. The resolution rule says that if we take these clauses as Input, then the Output is $C^1 \vee C^2$ (i.e. we simply "cancel" the clashing literals L and $\neg L$ and disjoin what remains). In the symbolism of a rule of inference, we have

$$\begin{array}{c} C^1 \vee L \\ \underline{C^2 \vee \neg L} \\ C^1 \vee C^2 \end{array}$$

This simple form is not adequate and a more general form is

needed involving certain substitutions called *most general unifiers (mgus)*. In this more general form, clashing literals are still cancelled, but after unification has brought them into syntactic identity. Furthermore, Robinson's algorithm for finding mgus can be implemented with reasonable efficiency, giving the means of feasibly constructing automated theorem provers. Such devices have been extensively examined in the context of mathematics, see [20], and a number of new results have been established in various areas in addition to verifying old results (in classical analysis for example). The drawback, however, in using resolution is that it involves an immense amount of searching for clashing literals and hence automated theorem provers using it run slowly in comparison with modern procedural languages such as PASCAL or C.

Interesting as these applications to mathematics are, they are a little peripheral to the main thrust of this work. Developments made by Colmerauer et al. in Marseilles, [10], Kowalski and van Emden in Imperial College, [6, 39], and Warren in Edinburgh and Manchester, [41], identified a significant fragment of first order predicate logic (the Horn clause subset) relative to which a restricted form of resolution (SLD-resolution) ran as fast as conventional languages. Note that by grouping all positive atoms in a clause to one end, and all negative ones to the other, we can write an arbitrary clause in the form $A_1 \vee A_2 \vee \ldots \vee A_m \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$. In turn this can be written as $A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$, where $\leftarrow$ denotes the connective "material implication." The relative slowness of resolution can now be traced to the presence, in general, of more than one atom in the "head" of this clause, that is, to $m > 1$, which causes a combinatorial explosion in search. Restricting syntax to allow only the case $m = 1$ results in so called (definite) program clauses of the type $A \leftarrow B_1, B_2, \ldots, B_n$, where A and all the B_i are atoms and the commas in the "body" denote conjunction. It also results in SLD-resolution running very fast. It is convenient to abuse notation and allow n to be zero to indicate that the body of a clause is empty, so that the clause in question is a *unit* clause $A \leftarrow$ or a "fact". This is in contrast to the conditional statement represented by a clause whose body is not empty.

Thus, in this paradigm (logic programming), a program is thought of as a finite set of axioms (each program clause being an axiom) in a formal system whose only rule of inference is SLD-resolution. Computation is thus (controlled) deduction. Moreover, soundness and completeness were both established for such systems and, in addition, it was shown that given any partial recursive function, there is a logic program that computes it. Thus, logic programming systems have as much power as conventional procedural languages despite the restricted syntax. This work led to the programming languages PROLOG and PARLOG (a parallel implementation), see [4, 19] for theoretical foundations and [9] for programming practice. Whilst not especially suited for numerical computation, logic programming languages are ideal for work in deductive databases, AI, and natural language processing in which first order predicate logic is viewed as a knowledge representation language. Current work which aims to incorporate λ-terms in clause bodies, and hence to amalgamate logic and functional programming styles, should result in increased flexibility. Here is an example of a PROLOG program (not quite in PROLOG syntax) which is a quick-sorting program intended to sort lists of non-negative integers and has two built-in predicates *le* and *gr*:

$$
\begin{aligned}
&qsort(nil, nil) \leftarrow\\
&qsort(H.T, S) \leftarrow part(H, T, P, Q), qsort(P, P1), qsort(Q, Q1),\\
&\qquad\qquad append(P1, H.Q1, S)\\
&part(R, H.T, H.X, Q) \leftarrow le(H, R), part(R, T, X, Q)\\
&part(R, H.T, X, H.Q) \leftarrow gr(H, R), part(R, T, X, Q)\\
&part(X, nil, nil, nil) \leftarrow\\
&append(nil, X, X) \leftarrow\\
&append(H.T, X, H.Y) \leftarrow append(T, X, Y)
\end{aligned}
$$

It should be observed that in addition to the operational semantics and the fixed point semantics (or *denotational semantics* as

it is often termed) that exist for procedural programs, logic programs P have a third semantics, the *declarative semantics*. This term simply refers to the model-theoretic meaning P has when viewed as a first order theory, namely, the set of all logical consequences of P. The completeness alluded to earlier, when properly formulated, says that the set M_P of those ground atoms (i.e. those atoms containing no variable symbols) which are derivable or provable from the clauses in P via SLD-resolution coincides precisely with the set of ground atoms which are logical consequences of P, that is, those ground atoms true in every model of P. Unfortunately, this set M_P (which can be thought of as the set of things which P computes) is not easy to get hold of when considered in these terms. It is at this point that the fixed point semantics of P enters the picture, for it is a major result of the theory, Theorem 4 below, that M_P coincides with the least fixed point of an operator T_P which can be associated with P and is analogous to the operator Φ discussed in Example 1; moreover T_P provides a relatively simple way of obtaining M_P. Again, we will consider these issues by reference to a simple example as follows.

Example 2 Consider the program P:

$$q(b) \leftarrow$$

$$q(s(y)) \leftarrow q(y)$$

$$p(s(s(x))) \leftarrow p(a), q(x)$$

which does not compute anything significant but is manageable and illustrates the main ideas. We start by observing that the underlying first order language $\mathcal{L}$ for this example contains just the following: constant symbols a, b; variable symbols x, y; a unary function symbol s; unary predicate symbols p, q. Let

$$U_P = \{s^m(a), s^n(b); m, n \in N\}$$

denote the set of all ground terms which can be formed using the symbols s, a, b, where s^n is informal shorthand for n occurrences of s; U_P is called the *Herbrand universe* for $\mathcal{L}$. Similarly, let

$$B_P = \{p(t), q(t'); t, t' \in U_P\}$$

denote the set of all ground atoms which can be formed using the symbols p, q and ground terms from U_P; B_P is called the *Herbrand base* for $\mathcal{L}$. There are canonical interpretations for $\mathcal{L}$, called *Herbrand interpretations*, which can be constructed out of the elements of $\mathcal{L}$ as follows: the constant symbols a, b in $\mathcal{L}$ are assigned to themselves in U_P; the mapping $U_P \to U_P$ defined by $t \mapsto s(t)$ of arity one is assigned to the unary function symbol s; since we are working in classical two valued logic, we assign a unary relation I^p on B_P (i.e. a subset of B_P) to the unary predicate symbol p and likewise a unary relation I^q to q to obtain the interpretation $I^p \cup I^q$. Since the assignment to constant symbols and function symbols is fixed, Herbrand interpretations I can be identified with subsets I of B_P by: ground atom $p(t)$ is *true* relative to I, written $I \models p(t)$, iff $p(t) \in I$; ground atom $q(t')$ is *true* relative to I, again written $I \models q(t')$, iff $q(t') \in I$. In this way, the set of all Herbrand interpretations for $\mathcal{L}$ can be identified with the power set $\mathcal{P}(B_P)$ which we will henceforth write as I_P. The set I_P we will regard as a complete lattice relative to the partial order of set inclusion whose bottom element is the empty set, and in which the infimum and supremum of an arbitrary collection of elements (subsets of B_P) are the intersection and union respectively of the collection. It is this complete lattice which is the domain of the operator T_P and is the usual domain on which fixed-point semantics is carried out for logic programs. Let us note therefore that, in general, this operator is defined by $T_P : I_P \longrightarrow I_P$ where

$$T_P(I) = \{A \in B_P;\ \text{there is a ground instance } A \leftarrow B_1, B_2, \ldots, B_n \text{ of a clause in } P \text{ satisfying} I \models B_1 \wedge B_2 \wedge \ldots \wedge B_n\}$$

(a ground instance of a program clause is simply a clause containing no variable symbols, so that elements of U_P have been assigned to each variable symbol). In practice, $T_P(I)$ is obtained by matching all the atoms in a given clause body with elements of I and collecting up corresponding clause heads. For example, with

$$I = \{p(a), p(s(a)), q(a), q(s(a)), q(b)\}$$

in our present example, we get

$$T_P(I) = \{p(s^2(a)), p(s^3(a)), p(s^2(b)), q(s(a)), q(s^2(a)), q(b), q(s(b))\}.$$

All the ideas presented in this example carry over completely to the case of an arbitrary program P in which all clauses are definite, a *definite* program. Now, a central fact emerges concerning T_P, *the immediate consequence operator* to give it its name, which is that T_P is *lattice-continuous* in the sense that $T_P(\sup(X_\alpha)) = \sup(T_P(X_\alpha))$ for every directed family (X_α) of subsets of I_P, where $\sup(X_\alpha)$ denotes the supremum of (X_α); this property of T_P is the analogue of recursiveness in the case of the operator Φ in Example 1. Once more, least fixed points of such operators play a fundamental role and the facts in brief are as follows. We inductively form the chain (I_n) in I_P by: $I_0 = \phi$ and $I_{n+1} = T_P(I_n)$, just as for Kleene's theorem, and take $\sup(I_n)$, which is often denoted $T_P \uparrow \omega$. This time we apply an abstract form of Kleene's first recursion theorem, due to Tarski, see [38] and also [18]. We obtain that $T_P \uparrow \omega$ is the least fixed point, lfp(T_P), and the following theorem due to van Emden and Kowalski, see [19, 39], shows the importance of this fixed point.

Theorem 4. *For any definite logic program P, we have $M_P = T_P \uparrow \omega = lfp(T_P)$.* ■

Carrying out the construction described above in the case of Example 2 shows that

$$M_P = T_P \uparrow \omega = \{q(b), q(s(b)), q(s^2(b)), \ldots\},$$

as is readily checked, and the elements of this set are exactly those ground atoms which P computes.

5. The topological viewpoint

Despite the fact that definite logic programs T_P can compute all computable functions, there is a need to extend syntax to make them more expressive. This means that we want to allow negative literals in the bodies of clauses (and hence arbitrary first order formulae). Once that is done, however, T_P fails to be monotonic and hence fails to be lattice-continuous (monotonicity is an easy consequence of lattice continuity of T_P or of recursiveness in the case of the operator Φ) so that the standard approach discussed in §4 does not apply. A partial remedy is to consider syntactic devices such as stratification and local stratification, see [5]. Nevertheless, the problem arises of finding fixed points of non-monotonic operators on I_P, and in this section we sketch methods being developed by us in [31, 32, 33] to solve this problem using topological notions and in particular quasi-metrics.

The following definition can be found in [27, 35].

Definition 3 Let X be a non-empty set. A *quasi-metric* on X is a map from $X \times X$ to the non-negative real numbers including $+\infty$ satisfying:
1. $d(x,x) = 0$;
2. $d(x,y) \leq d(x,z) + d(z,y)$;
3. if $d(x,y) = d(y,x) = 0$, then $x = y$.
A quasi-metric d is called an *ultra*-quasi-metric if it satisfies the strong triangle inequality
2′. $d(x,y) \leq \max\{d(x,z), d(z,y)\}$.

Notice that $d(x,y)$ and $d(y,x)$ are different in general. Quasi-metrics have been used in program semantics (for procedural programs) to reconcile the two standard approaches (Scott domains and metric spaces) to solving recursive domain equations. They can be viewed as categories enriched over the unit interval $[0,1]$, see [8], and this observation permits the development of many of their basic properties following ideas of W. Lawvere.

Given a quasi-metric d on X, there is an associated metric $d^\star$ defined on X by $d^\star(x,y) = \max\{d(x,y), d(y,x)\}$. One then says that (X,d) is *totally bounded* if the metric space $(X,d^\star)$ is totally bounded. Moreover, *d induces* a natural topology on X in which a set O is called *open* if, for every $x \in O$, some ϵ-ball $B(\epsilon,x)$ (where $B_\epsilon = \{y \in X; d(x,y) < \epsilon\}$) is contained in O.

The two examples which follow are taken from [35].

Example 3 Let $(D,\leq)$ be an arbitrary partially ordered set and define d on $D \times D$ by

$$d(x,y) = \begin{cases} 0 & \text{if } x \leq y; \\ 1 & \text{otherwise.} \end{cases}$$

Then d is an ultra-quasi-metric, called the *discrete quasi-metric*, which induces the Alexandroff topology and moreover (D,d) is totally bounded iff D is finite.

Example 4 Let $(D,\leq)$ be any Scott domain, let B_D denote the set of compact elements of D and let $r : B_D \to N$ be a map (a rank function) such that $r^{-1}(n)$ is a finite set for each $n \in N$. Define d on $D \times D$ by

$$d(x,y) = \inf\{2^{-n}; e \leq x \Rightarrow e \leq y \text{ holds for every } e \text{ of rank } \leq n\}.$$

Then d is an ultra-quasi-metric which induces the Scott topology of D and (D,d) is totally bounded.

Definition 4 A sequence (x_n) in the quasi-metric space (X,d) is said to be *forward Cauchy* if, for each $\epsilon > 0$, there is a natural number k such that $d(x_l, x_m) \leq \epsilon$ whenever $k \leq l \leq m$.

Definition 5 Let (x_n) be a forward Cauchy sequence in a quasi-metric space (X,d). A point $x \in X$ is a *Limit* of (x_n), written $x = \operatorname{Lim} x_n$, if, for every $y \in X$, we have $d(x,y) = \lim_{n\to\infty} d(x_n,y)$. The space X is said to be *complete* if every forward Cauchy sequence in X has a Limit.

The forward Cauchy property of the sequence (x_n) implies that the sequence $d(x_n,y)$ is itself Cauchy in the real line, so that the definition just given is meaningful. Moreover, Limits of forward Cauchy sequences are unique when they exist.

Definition 6 Let (X,d) be a quasi-metric space and suppose $f : X \to X$ is a mapping.
1. f is *non-expansive* if, for all $x, y \in X$, we have $d(f(x), f(y)) \leq d(x,y)$.
2. f is *contractive* if there exists a positive number $c < 1$ such that, for all $x,y \in X$, we have $d(f(x), f(y)) \leq c.d(x,y)$.
3. f is *Continuous* if, for all forward Cauchy sequences (x_n) and x in X, we have $\operatorname{Lim} f(x_n) = f(x)$ whenever $\operatorname{Lim} x_n = x$.

The following theorem is due to Jan Rutten, [27, Theorem 3.7].

Theorem 5. *Let (X,d) be a complete ultra-quasi-metric space and suppose $f : X \to X$ is non-expansive.*

1. If f is Continuous and there is an x in X with the property that $d(x, f(x)) = 0$, then f has a fixed point which is the least fixed point above x in the order $\leq_X$ defined by $y \leq_X z$ iff $d(y,z) = 0$.

2. If f is Continuous and contractive, then f has a unique fixed point. ■

Attempts to use the Banach contraction mapping theorem in logic programming have been made with some success in [12], and in [17] where problems arising out of attempts to formalize common sense reasoning are considered. Nevertheless, it is our claim that it is quasi-metrics that should be used instead, in conjunction with theorems such as Theorem 5. This point of view is substantiated by the following two observations: (i) The topology underlying the declarative semantics of definite programs is the Scott topology, see [32], which is not metrizable. This means that it is impossible to recover the classical theory of §4 with metrics. (ii) It is not usual for T_P to have unique fixed points (rather, the set of such forms a complete lattice) and this means that in general T_P is not a contraction relative to any metric. To finish this paper, we therefore briefly indicate how quasi-metrics can be used in logic programming.

First, consider an arbitrary definite logic program P and view I_P as a partially ordered set, under set inclusion, endowed with the discrete quasi-metric defined in Example 3. The following facts are established in [33]:

1) A sequence (I_n) in I_P is forward Cauchy iff it is eventually increasing.

2) (I_P, d) is complete.

3) The following are equivalent for any forward Cauchy sequence (I_n) in (I_P, d)

a) $\mathrm{Lim}\, I_n = I$.

b) $I_n \to I$ in the Scott topology and I is the greatest limit (in the Scott topology) of (I_n).

4) If (I_n) is a forward Cauchy sequence in (I_P, d), then $(T_P(I_n))$ is also a forward Cauchy sequence.

5) T_P is Continuous relative to d.

6) T_P is non-expansive relative to d.

Noting that the empty set ϕ satisfies $d(\phi, T_P(\phi)) = 0$, we can apply Rutten's theorem and, on examining its proof, we conclude that T_P has a fixed point equal to $\mathrm{Lim}\, T_P^n(\phi)$. This fixed point is, by Observation 3 above, equal to $gl(T_P^n(\phi))$ as defined in [32] and

this, in turn is equal to $\bigcup T_P^n(\phi)$, by [32, Proposition 8]. In this way, we recover the classical least fixed point of T_P and Theorem 4 follows immediately from this. ■

For our second and final application let P denote an arbitrary, not necessarily definite, program. This time we will think of I_P as a Scott domain (i.e. a bounded-complete ω-algebraic cpo) under set inclusion whose compact elements are the finite sets, the collection of which we will denote by B_D. We define a *level mapping* for P to be a function $l : B_P \to N$, and we assume that $l^{-1}(n)$ is a finite set for every n. Such mappings have found application in several places in logic programming including uses in defining stratification, in questions of termination of logic programs, and in treating completeness issues. Given a level mapping l we define the function $r : B_D \to N$ by $r(I) = \max_{A \in I}(l(A))$, for nonempty I, and set $r(\phi) = 0$. We will call r the *rank* function determined by l. Now let d denote the quasi-metric defined as in Example 4 so that (I_P, d) is complete and totally bounded, and d induces the Scott topology on I_P. The central facts we need, established in [33], concern the connection between quasi-metric notions and corresponding ones in the Cantor topology on I_P (this latter topology is denoted by Q in [32]) and are as follows:

1) For a sequence (I_n) in I_P and $I \in I_P$, the following statements are equivalent.

a) $I_n \to I$ in the topology Q.

b) (I_n) is forward Cauchy, $I_n \to I$ in the Scott topology and $I = gl(I_n)$, the greatest limit of (I_n).

2) Let (I_n) be a forward Cauchy sequence in (I_P, d). Then $\mathrm{Lim}\, I_n = I$ iff $I_n \to I$ in Q.

3) If T_P is non-expansive relative to d, then it is continuous in the topology Q.

4) T_P is Continuous relative to d iff it is continuous in the topology Q.

Once again it will be best to illustrate these ideas by considering them in relation to a simple example, as follows.

Example 5 Let P be the program:

$$p(o) \leftarrow$$

$$p(s(x)) \leftarrow \neg p(x)$$

This program is perhaps "intended" to compute the even natural numbers ($p(x)$ can be interpreted to mean "x is even", s can be interpreted to be the successor function). This program is not definite and is neither stratified nor locally stratified, so that the standard approach does not apply. Define the level mapping l on B_P by $l(p(s^n(o))) = n$ for each n. We note that in this case T_P is not non-expansive for if we take

$$I_1 = \{p(o), p(s(o))\}$$

and

$$I_2 = \{p(o), p(s(o)), p(s^2(o))\},$$

then

$$T_P(I_1) = \{p(o), p(s^3(o)), p(s^4(o)), p(s^5(o)), \ldots\}$$

and

$$T_P(I_2) = \{p(o), p(s^4(o)), p(s^5(o)), \ldots\}.$$

Thus, we have $d(I_1, I_2) = 0$ and yet $d(T_P(I_1), T_P(I_2)) = 2^{-2}$. Consider powers $I_n = T_P^n(\phi)$, the first few of which are as follows: $I_1 = B_P$, $I_2 = \{p(o)\}$, $I_3 = B_P \setminus \{p(s(o))\}$, $I_4 = \{p(o), p(s^2(o))\}$, $I_5 = B_P \setminus \{p(s(o)), p(s^3(o))\}$, etc. Using [33, Proposition 7] we obtain that $d(I_n, I_{n+1})$ takes value 0 if n is even and takes value 2^{-n+1} if n is odd. This is enough to show that the sequence (I_n) is forward Cauchy and therefore converges to I, say, in Q. By [32, Proposition 4] it is clear that (I_n) converges in Q to the set $\{p(o), p(s^2(o)), p(s^4(o)), \ldots\}$ which therefore coincides with I. Since T_P is continuous in Q by [32, Corollary 6], it follows that I is a fixed point of T_P by a simple argument using the uniqueness of limits in Q. Thus, the set I of "even natural numbers" is a model of P. In fact, it is not hard to see that I is the only fixed point of T_P. ■

References

[1] The Alvey Programme of Advanced Information Technology. Alvey Directorate: Millbank Tower, Millbank, London SW1P 4QV, U.K.

[2] Research in Logic for Information Technology: Logic for IT. Science and Engineering Research Council: Polaris House, North Star Avenue, Swindon, Wiltshire, U.K.

[3] S. Abramsky, D. M. Gabbay and T. S. E. Maibaum (eds.), Handbook of Logic in Computer Science, Volumes 1 to 6. Oxford Science Publications, Oxford University Press: Oxford, 1994.

[4] K. R. Apt, *Logic programming*, *in* Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, (ed.) J. van Leeuwen, 493-574. Elsevier Science Publishers: Amsterdam, 1990.

[5] K. R. Apt, H. A. Blair and A. Walker, *Towards a theory of declarative knowledge*, *in* Foundations of Deductive Databases and Logic Programming, (ed.) J. Minker, 89-148. Morgan Kaufmann Publishers Inc.: Los Altos, 1988.

[6] K. R. Apt and M. H. van Emden, *Contributions to the theory of logic programming*, J. ACM **29** (1982), 841-862.

[7] J. Barwise (ed.), Handbook of Mathematical Logic. North-Holland: Amsterdam, 1977.

[8] M. M. Bonsangue, F. van Breugel and J. J. J. M. Rutten, Generalized ultrametric spaces: completion, topology, and powerdomains via the Yoneda embedding, Technical Report CS-R9560, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[9] I. Bratko, Prolog Programming for Artificial Intelligence. International Computer Science Series, Addison-Wesley Publishing Company: 1988.

[10] A. Colmerauer, H. Kanoui, P. Roussel and R. Pasero, Un système de communication homme-machine en franąis, Groupe de recherche en intelligence artificielle, Université d'Aix-Marseille, 1973.

[11] N. Cutland, Computability: An Introduction to Recursive Function Theory. Cambridge University Press: Cambridge, 1980.

[12] M. C. Fitting, *Metric methods: three examples and a theorem*, J. Logic Programming **21** (1994), 113-127.

[13] D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, Volumes 1 to 5. Oxford Science Publications, Oxford University Press: Oxford, 1994.

[14] M. Gelfond and V. Lifschitz, *Classical negation in logic programs and disjunctive databases*, New Generation Computing **9** (1991), 365-385.

[15] Y. Gurevich, *The value, if any, of decidability*, Bull. European Association for Theoretical Computer Science **55** (1995), 129-135.

[16] T. C. Hurley, *Benefits and advantages of an integrated mathematics and computer science degree*, Irish Math. Soc. Bulletin **32** (1994), 63-72.

[17] M. A. Khamsi, V. Kreinovich and D. Misane, *A new method of proving the existence of answer sets for disjunctive logic programs: a metric fixed point theorem for multi-valued mappings*, *in* Proceedings of the Workshop on Logic Programming with Incomplete Information, Vancouver, B.C., Canada, October 1993, pp. 58-73.

[18] J-L. Lassez, V. L. Nguyen and E. A. Sonenberg, *Fixed point theorems and semantics: a folk tale*, Information Processing Letters **14** (1982), 112-116.

[19] J. W. Lloyd, Foundations of Logic Programming (second edition). Springer-Verlag: Berlin, 1988.

[20] D. W. Loveland, Automated Theorem Proving: A Logical Basis. North Holland: New York, 1978.

[21] P. Odifreddi, Classical Recursion Theory. North-Holland: Amsterdam, 1992.

[22] D. Park, *Fixpoint induction and proof of program properties*, *in* Machine Intelligence Volume 5, (eds.) B. Meltzer and D. Michie, 59-78. Edinburgh University Press: Edinburgh, 1970.

[23] L. C. Paulson, Logic and Computation. Cambridge Tracts in Theoretical Computer Science No. 2, Cambridge University Press: Cambridge, 1987.

[24] J. A. Robinson, *A machine oriented logic based on the resolution principle*, J. ACM **12** (1965), 23-41.

[25] J. A. Robinson, Logic, Form and Function: The Mechanization of Deductive Reasoning. Edinburgh University Press: 1979.

[26] H. Rogers, Theory of Recursive Functions and Effective Computability. MIT Press: Cambridge, Mass., 1987.

[27] J. J. M. M. Rutten, Elements of generalized ultrametric domain theory, Technical Report CS-R9507, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[28] D. S. Scott, *Data types as lattices*, SIAM J. Computing **5** (1976), 522-587.

[29] D. S. Scott, *Lectures on a mathematical theory of computation*, *in* Theoretical Foundations of Programming Methodology, (eds.) M. Broy and G. Schmidt, 145-292. Reidel: Dordrecht, 1982.

[30] D. S. Scott, *Domains for denotational semantics*, *in* Proceedings 9th International Colloquium on Automata, Languages and Programming, (eds.) M. Nielsen and E. M. Schmidt, 577-613. Lecture Notes in Computer Science Volume 140, Springer-Verlag: Berlin-Heidelberg-New York, 1982.

[31] A. K. Seda, *Some applications of general topology to the semantics of logic programs*, Bull. European Association for Theoretical Computer Science **52** (1994), 279-292.

[32] A. K. Seda, *Topology and the semantics of logic programs*, Fundamenta Informaticae **24** (1995), 359-386.

[33] A. K. Seda, Quasi-metrics and the semantics of logic programs, Fundamenta Informaticae, to appear.

[34] S. G. Shanker (ed.), Gödel's Theorem in Focus. Croom Helm Ltd.: Kent, 1988.

[35] M. B. Smyth, *Totally bounded spaces and compact ordered spaces as domains of computation*, *in* Topology and Category Theory in Computer Science, (eds.) G. M. Reed, A. W. Roscoe and R. F. Wachter, 207-229. Oxford University Press: Oxford, 1991.

[36] M. B. Smyth and G. D. Plotkin, *The category-theoretic solution of recursive domain equations*, SIAM J. Computing **11** (1982), 761-783.

[37] V. Stoltenberg-Hansen, I. Lindström and E. R. Griffor, Mathematical Theory of Domains. Cambridge Tracts in Theoretical Computer Science No. 22, Cambridge University Press: Cambridge, 1994.

[38] A. Tarski, *A lattice-theoretical fixed point theorem and its applications*, Pacific J. Math. **5** (1955), 285-309.

[39] M. H. van Emden and R. A. Kowalski, *The semantics of predicate logic as a programming language*, J. ACM **23** (1976), 733-742.

[40] J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, Volumes A & B. Elsevier Science Publishers: Amsterdam, 1990.

[41] D. H. D. Warren and F. C. N. Pereira, An efficient easily adaptable system for interpreting natural language queries, DAI Research Paper No. 155, Department of Artificial Intelligence, University of Edinburgh, 1981.

[42] J. Woodcock and M. Loomes, Software Engineering Mathematics. Pitman Publishing: London, 1988.

A. K. Seda
Department of Mathematics,
University College,
Cork,
Ireland.
email:aks@bureau.ucc.ie

Book Review

Groups '93
Galway/ St Andrews

London Mathematical Society Lecture Note Series,
vols. 211 & 212

Ed. by C. M. Campbell, T. C. Hurley, E. F. Robertson,
S. J. Tobin & J. J. Ward

Cambridge University Press 1995
xii+304pp (vol. 1), xii+305pp (vol. 2)
ISBN 0-521-47749-2 (vol. 1), 0-521-47750-6 (vol. 2)

Reviewed by Rod Gow

The volumes under review contain selected papers from a conference on group theory held at University College Galway during the period 1-14 August 1993. This conference was the continuation of a series of conferences on group theory held at the University of St Andrews in 1981, 1985 and 1989, with the next conference to be held in Bath in 1997. There were 285 participants at the conference, with numerous principal lectures, invited lectures, research talks and workshops on computational group theory to entertain them.

It seems to the reviewer that large scale conferences devoted to a rather broad theme are less common these days than once they were. In group theory, conferences on groups of Lie type, representation theory of algebraic and related finite groups, groups and geometry, or computational group theory are dominant. This probably reflects the fact that researchers' interests are more narrowly focused on their specialities and they may imagine that there is a better chance of a pay-off in terms of a publication by attending conferences offering a concentrated diet of specialized material. Looking through the papers under review, I noticed that many topics popular 25 years ago are no longer represented. These include finite simple groups, ordinary character theory and